Universidad Carlos III de Madrid
Escuela Politécnica Superior

Doctoral Thesis

Enhancing the programmability and energy efficiency of storage in HPC and virtualized environments

Author: Pablo Llopis Sanmillán
Advisors: Dr. Francisco Javier García Blas
Dr. Florin Isaila

Computer Science and Engineering Department
Leganés, May 2016

Universidad Carlos III de Madrid
Escuela Politécnica Superior

Tesis Doctoral

Enhancing the programmability and energy efficiency of storage in HPC and virtualized environments

Author: Pablo Llopis Sanmillán
Advisors: Dr. Francisco Javier García Blas
Dr. Florin Isaila

Departamento de Informática
Leganés, May 2016

TESIS DOCTORAL

Enhancing the programmability and energy efficiency of storage in HPC and virtualized environments

AUTOR: **Pablo Llopis Sanmillán**
DIRECTORES: **Francisco Javier García Blas**
Florin Isaila

TRIBUNAL CALIFICADOR

PRESIDENTE:

VOCAL:

VOCAL:

VOCAL:

SECRETARIO:

CALIFICACION:

Leganés, a de de 2016

A mis padres.

Agradecimientos

TODO

Abstract

A decade ago computing systems hit a clock and power ceiling that places the energetic challenge among the most relevant issues in High Performance Computing (HPC). Motivated by the fact that computation is increasingly becoming cheaper than data movement in terms of power, our work studies and optimizes data movement across different levels of the software stack. We propose novel methodologies for analyzing, modeling, and optimizing the energy efficiency of data movement. More precisely, we propose methodologies to enhance the understanding of power consumption in the software I/O stack, and optimize the I/O energy efficiency in the operating system's I/O stack, low-level CPU device drivers, and virtualized environments. Our experimental results show that through the understanding of the different operating system layers and their interaction, it is possible to develop novel coordination techniques that optimize the energy consumption and increase performance of I/O workloads.

First, we develop a methodology for data collection, power and performance characterization, and modeling power usage in the I/O stack. Our work presents a detailed study of power and energy usage across all system components during various I/O-intensive workloads. We propose a data gathering methodology that combines software and hardware-based instrumentation in order to study I/O data movement, and develop novel power prediction models employing data analysis techniques.

Second, this thesis presents novel CPU-level optimizations that improve the energy efficiency of I/O workloads. We address two issues present in modern processors: thermal imbalance causing performance variation and an inefficient use of CPU resources during I/O workloads. We develop novel techniques for power optimization and thermal efficiency through cross-layer coordination of CPU and I/O management.

Third, we also focus on optimizing data sharing among virtual domains. In our work we refer to this as virtualized data sharing, which mainly differs from existing solutions by coordinating data flows through the software I/O stack. We develop a virtualized data sharing solution in order to reduce data movement among virtual environments, introducing new abstractions and mechanisms to more efficiently coordinate storage I/O.

Resumen

Hace una década, los computadores alcanzaron el límite físico de la frecuencia y potencia disipada, estableciendo el consumo energético como uno de los principales obstáculos en el campo de la computación de alto rendimiento. Motivados por el hecho de que la computación resulta cada vez menos costosa que el movimiento de datos en términos de energía, nuestro trabajo estudia y optimiza el movimiento de datos en varios niveles de la arquitectura software. En este trabajo proponemos nuevas metodologías para analizar, modelar y optimizar la eficiencia energética del movimiento de datos. Concretamente, proponemos metodologías para mejorar el análisis del consumo de potencia en la arquitectura software de E/S, así como optimizar la eficiencia energética de: la pila de E/S del sistema operativo, contoladores de la CPU y entornos virtuales de E/S. Los resultados experimentales muestran que, mediante la comprensión de la interacción de las capas del sistema operativo, es posible desarrollar nuevas técnicas de coordinacion que optimicen el consumo energético e incrementen el rendimiento de las cargas de trabajo de E/S.

En primer lugar desarrollamos una metodología para la recolección de datos y la caracterización del rendimiento y consumo de potencia en la pila de E/S. Nuestro trabajo presenta un estudio detallado del consumo energético y de potencia de cada uno de los componentes del sistema durante la ejecución de cargas de trabajo de E/S. Concretamente proponemos una metodología de captura de datos que combina instrumentación hardware y software para estudiar el movimiento de datos, con el fin de desarrollar nuevos modelos de predicción de consumo empleando técnicas de análisis de datos.

En segundo lugar, esta Tesis Doctoral presentamos nuevas optimizaciones a nivel de CPU que mejoran la eficiencia energética de las cargas de trabajo de E/S. Para ello consideramos dos problemas fundamentales en los procesadores modernos: el desequilibrio térmico que causa variablidad de rendimiento y el uso ineficiente de los recursos de la CPU durante cargas de trabajo de E/S. Además desarrollamos nuevas técnicas que optimizan la eficiencia energética a través de la coordinación entre las distintas capas del sistemas operativo que gestionan CPU y la E/S.

En tercer lugar, también centramos este trabajo en la optimización del intercambio de datos entre dominios virtuales. En nuestro trabajo nos referimos a esto como el intercambio de datos virtualizado, que se diferencia principalmente de las soluciones existentes mediante la coordinación de los flujos de datos mediante la cooperación entre distintos dominios virtuales. Para ello desarrollamos una solución de intercambio de datos que minimiza la copia de datos con el fin de reducir

el movimiento de datos, e introducimos nuevas abstracciones y mecanismos para coordinar de manera más eficiente el almacenamiento de E/S en entornos virtuales.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The number of operations per time unit has commonly been used as a metric for estimating anything from algorithm run-time to performance. Reducing the number of operations to be performed by a micro-processor has generally been associated with performance, power consumption, and run-time improvements. However, recent developments in manufacturing processes and limits on the amount of dissipated power have brought a paradigm change in which computation is less costly than transporting data across buses and silicon. As such, data movement is becoming a relevant concept present in computing systems of all scales. In fact, power and thermal constraints are becoming the main challenges for developing Exascale and micro-processors of the future.

1.1 Motivation

Traditionally, performance improvements have been the main focus for high-end computing systems. Modern scientific discoveries have been driven by an insatiable demand for high performance computing (HPC). However, power consumption has become an increasingly relevant issue since the power wall was hit about a decade ago. As we progress on the road to Exascale systems, energy consumption becomes a primary obstacle in the design and maintenance of HPC facilities. As of 2016, a simple extrapolation shows that an Exascale platform based on the most energy efficient hardware currently available in the Green500 [Gre15b] would consume over 140 MW at 7 GLOPS/Watt. However, the desirable goal has been set by the Department of Energy (DOE) to 20 MW [DoE14], meaning that this system would still exceed this limit by a factor of seven, thus turning it economically unfeasible due to its projected Total Cost of Ownership (TCO). Indeed, such systems will need to reach an energy efficiency of approximately 50 GFLOPS/Watt to face the Exascale challenge. Due to the key role of power constraints, future Exascale systems are expected to work

with a limited power budget, and be able to allocate power to different subsystems dynamically. In this scenario, the capability of predicting power consumption based on data movement and I/O operations is a useful resource. Actually, hardware vendors are already trying to provide more energy efficient components, and software developers are gradually increasing power-awareness in the current software stack, from applications to operating systems. For example, recent advances in processor technologies have enabled operating systems to leverage new energy efficient mechanisms such as DVFS (Dynamic Voltage and Frequency Scaling) or DCT (Dynamic Concurrency Throttling) in order to bound power consumption of computing systems.

In addition to the scaling problem, the economic and ecological cost of powering computing infrastructures are also a big concern. Several research studies have shown that power bills can contribute to as much as 50% of the TCO of a data center [JS08]. Indeed, the cost of powering a server approaches the acquisition cost of the hardware itself [VSS+10, Bar05]. In order to put this high energy usage into perspective, data centers in the U.S. alone were found to represent about 2% of the electricity consumption in 2010 [Koo11], which is roughly equivalent to the aviation industry. That same year, U.S. data centers consumed close to 80 billion kilowatt-hours of energy [Koo11]; this can be estimated to a total cost of about $5.9 billion[1]. In fact, this trend has been growing. U.S. data centers reportedly consumed an estimated 91 billion kilowatt-hours of electricity in 2013, the equivalent of 34 large coal-fired power plants, costing $6.7 billion and polluting around 100 MMTCO2[2]. If this growing trend continues, annual consumption is projected to increase by 2020 to 138 billion kilowatt-hours of energy, a cost over $10 billion, and pollute 150 MMTCO2 [Nat14].

The energy crisis of the last years and the ever increasing conscience about the negative effects of energy waste on the climate change have brought the sustainability both into public attention and under industry and scientific scrutiny. Green, or sustainable computing, has become the focus attention of initiatives like Green Grid [Gre15a], a global consortium dedicated to advancing energy efficiency in data centers and business computing ecosystems. This consortium has even established a metric called Power Usage Effectiveness (PUE) [BRPC08], which aims to measure and compare the power efficiency of data centers. As demonstrated by the successful emergence of the Green500 [Gre15b] list, which provides a ranking of the most energy-efficient supercomputers in the world, energy has become as significant as performance. Consequently, the performance-per-watt has been established as a new metric to evaluate supercomputers.

While many aspects need to be improved in the context of energy efficiency, including low-power electronics, memory, CPU, applications, cooling, and the data center itself, data movement was found to be one of the biggest energy-related chal-

[1]The latest cost estimation is known to be $4.5 billion for a 61 billion kwh consumption in 2006 [B+08]. The $6 billion figure is an extrapolation using the last known figure of 80 billion kwh [Koo11].

[2]Million metric tons of carbon dioxide is the measurement unit used for describing the magnitude of greenhouse gas emissions or reductions.

lenges [KBB+08, DoE14]. In addition, storage systems are expected to play an increasingly important role because the world's digital data grows exponentially, doubling every year, as revealed by a recent study conducted by the IDC [GR11]. The growth in size of data sets and the need to process and archive a large amount of information has resulted in a trend known as *Big Data*. And as the power cost of computation decreases, the cost of data movement increasingly becomes a more relevant issue [BC11]. The low performance of the I/O operations continues to present a formidable obstacle to reaching Exascale computing in the future large-scale systems, especially in I/O-intensive scientific domains and simulations. CPU speed and storage capacity are boosted by factors of approximately 400 and 100 every 10 years, respectively. Storage devices speed, however, develops at a much slower pace and increases only by a factor of 20 over the same time span.

This results in a special interest in optimizing I/O, and motivates the need for more research aimed at improving the energy efficiency of storage technologies. In spite of this, there is no sufficient research that studies the energetic impact of Input/Output (I/O) and data movement. While data movement and energy efficiency are relevant issues from large-scale domains to the micro-processor, this thesis targets one computing node and focuses on various operating system layers.

Our **hypothesis** is that inefficiencies related to data movement are present in different layers of the operating system. Based on performance metrics from various layers, we expect to understand power usage of the I/O stack and develop prediction mechanisms that can be used for estimating energy consumption of I/O workloads. Understanding the relationship between data movement and energy efficiency can be substantially improved by modeling the I/O accesses based on a series of runtime metrics from several I/O stack layers. Additionally, we expect that the energy efficiency associated with the data movement can be improved by techniques jointly optimizing for CPU utilization and I/O stack. Finally, we postulate that there is a need for novel abstractions and techniques for data movement in virtualized environments targeting to optimize the energy efficiency.

1.2 Objectives

The main objective of this thesis is to investigate, understand, and optimize power consumption related to data movement and propose new mechanisms that improve the energy consumption across various software layers. We break down the main objective into the following objectives.

- ***Understand*, *characterize*, and *model* energy consumed by data movement**. We will address this objective by investigating energy consumption across the I/O stack. To study energy consumption, we develop a methodology for gathering relevant data, and use data exploration and data analysis techniques to reveal how power is used in the software I/O stack. Leveraging the results of our analysis, we investigate predictive power consumption models for various I/O access patterns and workloads.

- **Explore novel CPU-level techniques for optimizing the performance and energy efficiency of data movement in the node-level I/O stack.** We will address this objective by investigating whether it is possible to improve I/O efficiency on micro-processor levels through a software approach. To achieve this objective, we explore methodologies for increasing cooperation between operating system layers that can be leveraged to improve I/O efficiency. As a result, we expect to be able to develop power- and thermal-aware software that makes intelligent use of CPU resources to improve CPU energy and power consumption.

- **Research novel approaches for enhancing performance and energy efficiency of data sharing across virtualized domains**. We address this objective by investigating novel abstractions and mechanisms that improve data sharing between virtual domains. We aim to study intra-domain coordination mechanisms that increase the efficiency of data sharing operations.

1.3 Overview

The purpose of this chapter is to present an overview that provides an holistic description of the work introduced in this thesis. Our work studies and optimizes data movement across different levels of the operating system's I/O stack. More precisely, we propose contributions to enhance the understanding of power consumption in the software I/O stack, and optimize the I/O energy efficiency in virtualized environments, the operating system's I/O stack, and low-level CPU device drivers, as depicted in Figure 1.1. Our contributions show that through the understanding of the different operating system layers and their interaction, it is possible to develop coordination techniques that optimize the energy consumption and increase performance of I/O workloads.

To address our first objective, we develop a methodology for data collection, power and performance characterization, and modeling power usage in the I/O stack. We perform a detailed study of power and energy usage across all system components during various I/O-intensive workloads. To perform an exhaustive examination, our work combines software and hardware-based instrumentation in order to study I/O data movement through exploratory data analysis. This data-driven process reveals detailed knowledge about how the system shifts between different power and performance regimes, and which layers and algorithms of the I/O stack are responsible. As a result of our analysis and characterization, we provide I/O power models that are able to predict power consumption of I/O workloads that perform various access patterns. This work is presented in Chapter 3.

For addressing the second objective of this thesis, we focus on the CPU and its impact on I/O energy efficiency, motivated by the fact that it is one of the most power-hungry components in a system. We examine the behavior of the CPU under I/O intensive workloads, and make two observations. First, we learn that in spite of being the most power-proportional component, the CPU is agnostic to external events such as heat or I/O activity. Second, we note that there is a thermal imbal-

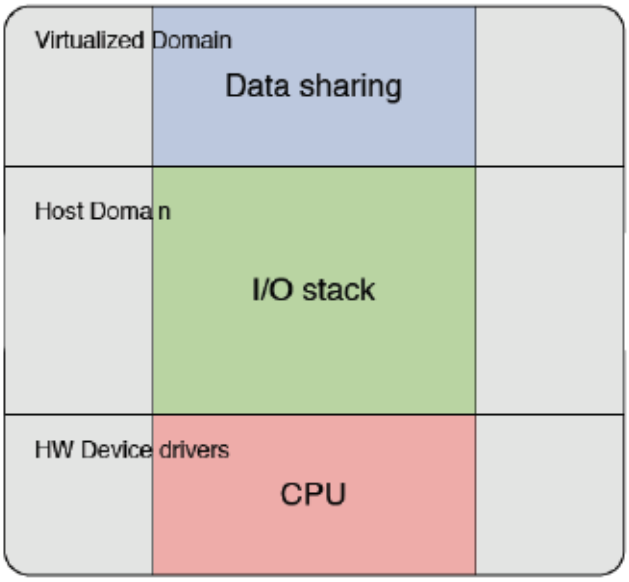


Figure 1.1: *The contributions of this thesis span multiple levels of the software I/O stack.*

ance that causes the CPU behave like a heterogeneous system. For each of these two cases, we develop runtimes that are able to decrease energy consumption for I/O workloads. Motivated by our first observation, we develop I/O-aware performance state selection. We are able to detect I/O regimes and shift power states accordingly in order to lower CPU power usage without reducing performance. Our second observation motivates us to develop thermal and I/O-aware thread placement, where computationally intensive and I/O intensive workload threads are placed in a thermal-aware fashion to optimize CPU power consumption. This work is presented in Chapter 4

Finally, we address our third objective by optimizing energy efficiency based on techniques that minimize data movement through improved data sharing in virtualized environments. We focus on optimizing data sharing between co-hosted virtual machines. In our work we refer to this as *intra-domain data sharing*, which mainly differs from existing solutions in the way the data moves across the software I/O stack. We develop virtualized data sharing (VIDAS) in order to reduce data movement across virtual environments. VIDAS proposes new abstractions and mechanisms to coordinate storage I/O across virtual domains more efficiently, reduce data movement by creating intra-domain shared access spaces, relax POSIX consistency to allow flexible data write and update policies, and expose data locality. We argue that these abstractions and mechanisms can be used to build an efficient para-virtualized file system, and demonstrate reduced energy consumption and increased performance for various collective I/O access patterns. This work is presented in Chapter 5.

1.4 Structure and Contents

The remainder of this document is structured in the following way.

- Chapter 2 *State of the art* contains existing research on the topics of power, energy efficiency, storage, and virtualization.
- Chapter 3 *Analysis and modeling of energy consumption in the I/O stack* details the analysis of power consumption caused by data movement across the I/O stack. We also present analytical models that are capable of predicting power and energy consumption for various I/O workloads and access patterns.
- Chapter 4 *CPU-level data I/O energy efficiency and optimizations* demonstrates CPU power, thermal, and performance improvements through software that performs intelligent low-level performance management.
- Chapter 5 *Virtualized I/O and data sharing* presents virtualized data sharing solutions that improve performance and power consumption of various data sharing operations among virtual domains.
- Chapter 6 *Conclusions* presents a summary of the contributions and results obtained in this thesis. We also provide insights into future challenges and outline directions for future research.

Chapter 2

State of the Art

In the last years, the landscape of High Performance Computing (HPC) has changed significantly to address two important challenges. First, the challenge of increasing the energy efficiency to improve the amount of computation that can be achieved with a limited power budget. Second, the exponential growth of data sets requires a growing amount of storage and data processing systems. These two challenges are related because data movement is becoming costlier than computation in terms of power. An added challenge is that many of these systems are moving to the cloud due to the economical advantage and flexibility provided by a virtual infrastructure. Moving workloads to a virtualized domain prompts the question of how efficiently data is moved through the node's software stack.

In this chapter, we explore the state of the art of these topics. We start introducing concepts and classifications of storage systems that are related to power solutions detailed in other sections of this chapter. In the following sections we focus on the topics of power and energy. We detail power measurement methodologies, and detail node-level solutions for improving energy efficiency. We also analyze distributed storage techniques that address power proportionality. While this thesis studies and proposes solutions focusing on one node, we also provide state of the art works on distributed storage techniques, since our future goal would be to apply our work in distributed environments. Finally, we also describe storage virtualization techniques, laying out virtualized data sharing solutions and inter-domain data sharing mechanisms.

2.1 Storage system types

In this section different types of storage systems are described and classified, guiding the reader in a top-down fashion from higher-level concepts to lower-level, single-component storage types. This synthesis is important because it will show the reader

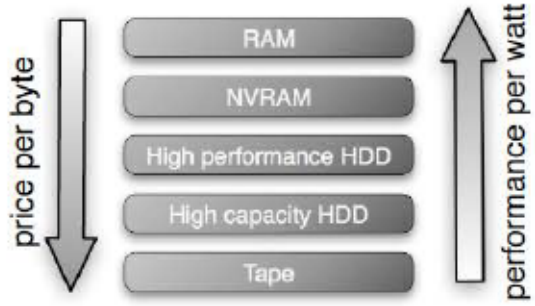


Figure 2.1: *Storage media can be organized in a hierarchy where trade-offs for dollar per watt and dollar per byte exist at opposite ends of the spectrum.*

about the different storage technologies, and how each of them relates to different power-aware solutions.

NVRAM: Non-volatile random access memories bridge the performance gap between fast, volatile RAM and a traditional disk-based storage, which is orders of magnitude slower. For this reason, NVRAM devices are emerging as disruptive storage for high-performance computing systems [CCM+10]. There are different types of NVRAM technologies, like phase-change memories (PCM) and spin-torque transfer memories (STTM) among others. Flash-based solid state drives (SSDs) are the most popular of the existing non-volatile RAM devices. They are more expensive and have lower capacities than HDDs, but offer greater performance, lower latencies through faster access times, and better energy efficiency in return. Their characteristics make them a popular choice as persistent caches, and some techniques taking advantage of this fact are described in Section 2.5.

HDD: Hard disk drives are the most common of storage devices. They consist of magnetic rotating disks. A low price-per-byte ratio, combined with high capacity and relatively good performance makes them a popular choice for most storage systems. Their effectiveness as energy efficient storage devices and drawbacks are detailed in Section 2.5.

RAID: A redundant array of independent disks (RAID) offers a logical storage unit which consists of individual disk drives. Different RAID configurations, numbered from 0 to 6, provide different parity-based redundancy mechanisms. Table 2.1 offers a comparative overview of different RAID levels and their energetic cost overhead. The energetic overhead is counted as the additional number of disks which are required for the increased storage space dedicated to either mirroring or parity data. Traditional RAID storage systems are not power proportional because the whole storage system needs to be powered up regardless of the workload. There are, however, solutions that have been proposed to make RAID more power proportional and energy efficient [LW04]. These solutions are detailed in Sections 2.6 and 2.5.1, respectively.

SAN and NAS: Network attached storage (NAS) refers to storage systems which

RAID Level	Fault tolerance	Space efficiency	Energetic overhead
RAID 0	0	1	0
RAID 1	$n-1$	$1/n$	$n-1$
RAID 5	1	$n-1$	1
RAID 6	2	$n-2$	2

Table 2.1: *Comparative summary which shows trade-offs between fault tolerance, space efficiency, and energetic overhead for different RAID levels. n represents the number of disks which compose the disk array.*

can be accessed over the network. Storage area networks (SAN) are networks dedicated to storage, offering block-level access to multiple disks. A typical example of a SAN is Fibre Channel. Many possible designs and implementations of SAN and NAS are possible, and therefore no direct relation between networked storage systems and power proportionality (or lack thereof) can be established without additional information. However, due to the fact that networked storage is an essential part of the power proportionality of a large-scale storage system, especially their data layout strategy, Section 2.6.1 has been dedicated to this matter.

Tape: Magnetic tape data storage offers the highest capacity for the lowest price, but suffers from poor performance and poor energy efficiency. The reliability, combined with the capacity and price benefits, makes tape an affordable solution for data archival.

Due to the fact that different storage devices have different properties (energy efficiency, capacity, price, performance, volatility), they are used together in order to combine their benefits, creating a *storage hierarchy*. This is especially the case for large data sets such as massive storage archives, where different tiers for low-cost and high-cost are established. In certain situations, the same kind of storage devices may be used at different tiers, such as high performance HDDs for one tier and high-capacity (and lower performance) HDDs for another tier [SNI12]. In the storage hierarchy there is a trade-off between capacity, performance, price, and power. As depicted in Figure 2.1, drives which have a higher capacity density usually offer a lower dollar per byte ratio, and are also usually slower than the media at the other end of the spectrum. The opposite is true for the performance per watt ratio. One popular configuration is to use non-volatile RAM (NVRAM) devices such as flash-based SSDs for frequently accessed data, while leaving unpopular content to slower, higher capacity hard disks or even tape [SNI12].

2.2 Power and energy

This section gives a detailed definition of the terms *power* and *energy*. Energy is the total amount of work a system has performed in a given time and is measured in joules (J) or watt-hours (Wh). In the context of this thesis, work is the amount of

electric charge transferred through the circuit. Power is the rate at which a system performs work. A system that performs E joules of work in a time frame of T seconds is said to deliver a power of $P = E/T$. Power is measured in watts (W). Although referred to simply as power, computer components specifically consume *electric power*. A system's power supply unit performs the conversion from AC current to DC current, which is then fed to the motherboard and individual components. The flow of electric charge through a circuit, the electric current, is also known as current intensity and is denoted as I. The electric current is measured in amperes (A) and represents the amount of electric charge that is transferred by the circuit per second. On direct current circuits, the electric power is mathematically represented as $P = IV$.

For a given component with a fixed voltage input V, the electric current can vary, resulting in variable power usage. For example, CPUs can greatly reduce their power usage by lowering their operating frequency. Similarly, disks can switch to low-power states where part of their circuitry is powered off and the disk platters remain spun down. This is the motivation behind some fine-grained power measurement techniques that are described in detail in Section 2.3.1, some of which perform current measurement for those wires that power specific system components. Power usage is then derived by multiplying the measured current with the line voltage.

2.3 Energy measurement methodologies

This section details how different research works deal with the problem of estimating power consumption in order to measure the effectiveness of their respective solutions. In some cases, power usage is not only measured, but also matched against captured traces. This allows to correlate performance with power consumption. For example, this can be helpful for checking the degree of power proportionality for storage systems. Many different approaches are chosen in order to measure power consumption. Some are more intrusive than others, ranging from hardware devices that measure power consumption of a whole system or some of its components, to modeling and tracing frameworks that estimate power usage based on different criteria. A taxonomy of power measurement methods is presented in Figure 2.2. The different measurement methods are reviewed in more detail below.

2.3.1 Power measurement devices

Hardware devices that measure power consumption are helpful for determining energy usage accurately. Due to the fact that power meters are expensive and their deployment is intrusive, scaling this approach even to small clusters can prove to be prohibitive. However, this is the most reliable way of measuring power consumption. This is especially interesting for validating power models and estimations performed through less intrusive methods that are able to scale-out with more ease. Several of these models are described in the following subsections.

There are different brands and kinds of power meters. A popular hardware

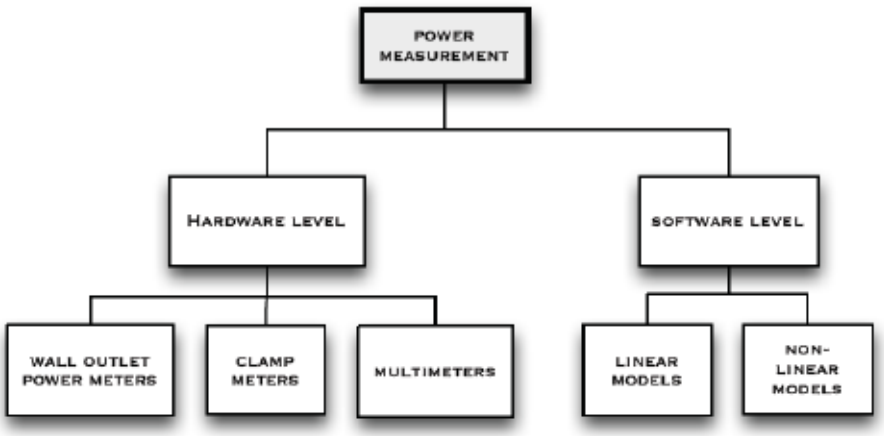


Figure 2.2: *Taxonomy of power measurement methods.*

tool for measuring energy usage is "Watts up? Power Meter" [SKZ08, VKUR10, CAKLR11]. This measurement device fits into standard AC outlets and appliances are plugged into the meter directly. Power meters poll at only a few Hertz, usually 1-4Hz. Because these kinds of devices sense AC power and power supplies are highly inefficient (10-30% of power is lost just in AC-DC conversion), power consumption measurement is not very precise. One of the drawbacks of these devices is that they are not suitable for measuring individual components. However, their deployment is relatively simple because of the "plug and play" approach.

Other devices like multimeters are directly attached to different points in the motherboards, providing power usage by component. Multimeters perform DC power sensing, achieving high precision measurements. Data can be sampled at much higher frequencies, in the order of hundreds of kHz, allowing for very fine-grained measurement and sampling. Because the multimeter attaches directly to the power supply unit's DC power lines, their deployment is very intrusive when compared to other approaches. In addition, multimeter devices are significantly more expensive than regular AC power meters.

A good example that takes advantage of such devices is the one described by Ge et al [GFS+10], which consists of a multimeter device that measures a sensor resistor connected to the circuit board through ATX extension cables. Data are polled at 4Hz and extracted through a serial port to an external system called PowerPack, whose purpose is to collect data. Figure 2.3 shows the setup configuration for conducting experiments. The authors are able to isolate power consumption by component. Ten power points are measured independently using 10 different multimeters attached at different points between the power supply and each component. The experiment is carried out on ATX boards. PowerPack combines data gathered by power meters together with performance data in order to generate power profiles for applications. In this configuration, each node component is measured in a different way. While the CPU fan, disks, and CPU are directly measurable by pins on the main power

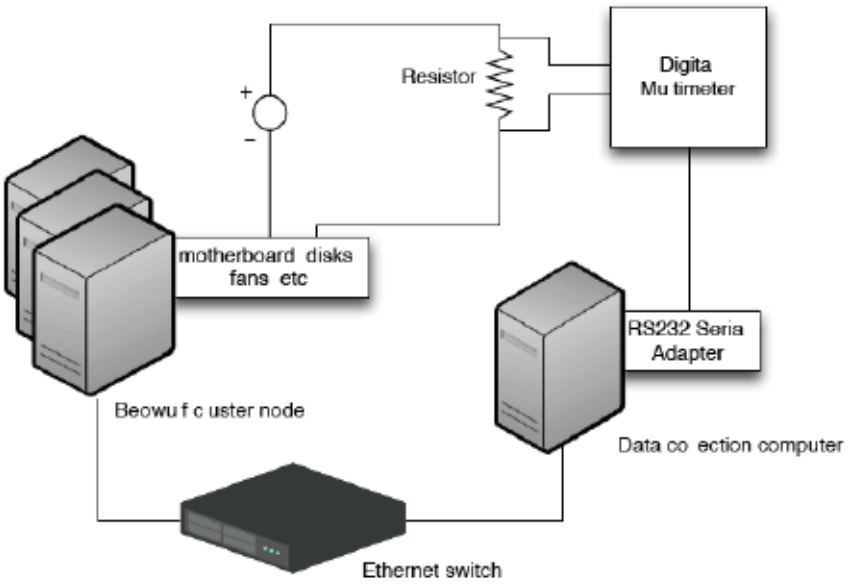


Figure 2.3: *PowerPack multimeter setup described in [GFS+10].*

connector, other components' power consumption can be experimentally measured by extrapolation. For measuring memory and network card consumption, the aggregate node power is measured with and without this node's component plugged in. The obtained results are compared to the vendor specifications in order to perform further verification.

A third kind of devices used for physically measuring power consumption that sits between AC power meters and DC multimeters are power clamps. Also known as clamp meters, these devices are able to measure AC and DC current without the need of cable manipulation or even unplugging, avoiding physical contact. This is especially useful for measuring isolated component power consumption by measuring the current that flows through the power supply unit's DC outputs, and is less intrusive than multimeter devices. Often, clamp meters do not provide an output interface other than an analog meter or digital screen, whereas other types of power meters usually include a serial interface that can be hooked up to a data collecting device, resulting in a semi-automated data capturing solution. Furthermore, clamp meters suffer from lower precision readings, although the errors are usually only a few hundred milliamps at most. Due to the fact that low current measurements are less reliable, cables can be winded around the clamp several times in order to increase reading resolution and achieve higher precision. In these cases, the obtained reading should be divided by the number of times the cable was wrapped around the clamp meter's loop. Tsirogiannis et al. [THS10] employ this technique in order to obtain isolated readings of hard disks, solid state disks, and CPUs.

2.3.2 Simulators and tracing tools

Many researchers rely on traces that have been captured on production systems with real workloads. Those traces are then replayed through simulators in order to estimate power consumption and test new power-aware storage strategies. For example, a team of Microsoft researchers gathers live traces that have been previously captured from a production environments such as Hotmail and Messenger, and replays them in simulations in order to predict energy savings. Power consumption is simply estimated by the percentage of servers of the baseline systems that are powered on [TDN11].

Q. Zhu and Y. Zhou [ZZ05] gather their energy usage from a disk simulator called DiskSim [BSSG08], which they have extended with a storage cache simulator called CacheSim. Energy estimations are done by analyzing cache misses and the power management scheme. Also, both idle and active times are taken into account in order to estimate energy through simulations with traces as input data. Power consumption deviations are reported to be under 2%.

In order to simulate energy consumption of GreenHDFS, the authors feed their trace-driven simulator with data extracted from the data sheet of system components [KB10]. Similarly, the approach described by Narayanan et al. [NDR08] is based on feeding simulators and testbeds with power data stemming from production manuals and traces gathered in production systems. Disk-level power consumption is measured in this way as well.

Another work focuses on evaluating the energy efficiency of existing MapReduce implementations. Chen et al. [CGF+10] propose a framework that addresses the problem of reproducing the hardware and software configuration of a production cluster under different, non-production environments. This problem is solved by generating workloads that effectively replay the statistical properties represented in the production traces. Using this framework, the authors are able to identify which workload configurations achieve the best performance and energy efficiency. According to the authors of this paper, having more and better production traces of different MapReduce workloads would yield data-driven and informed decisions that improve the design of other MapReduce-based systems in aspects such as scale, scheduling strategies, and system configuration.

M. Allalouf et al. [AAF+09] contribute a power modeling framework called STAMP. The practical use of this framework is threefold. First, it offers a way to estimate online power consumption for storage systems. Second, the framework can be used as an online power-aware capacity planning tool. Statistical performance information represents the host storage workload which is used as input for the storage models. Finally, the use of statistical information allows the framework to be employed for power and performance estimation on non-alive systems still in the design stage as well. Experimental results show how this modeling method achieves a reasonable error deviation of 2% to 9%, depending on the workload transfer size. The interest in this contribution stems from the fact that this work focuses on modeling based on workload and storage configurations. While other works tend to treat most I/O operations equally, considering disk utilization as a function of number of oper-

ations and time duration, STAMP is more accurate because more detailed workload characteristics are taken into account. As the authors demonstrate, STAMP can better exploit scenarios where trade-offs between performance and power exists in order to make systems more power-aware. Additional details about STAMP are described in Section 2.3.3.

M. Knobloch et al. [KMM12] analyze power performance of MPI applications. Their study is specially focused on energy saving opportunities that originate from busy-wait states that are shown to be highly power-inefficient. They extend the Scalasca [GWW+10] (Scalable analysis of large-scale applications) tool-set, which is designed to detect wait states, focusing on energy efficiency. It is examined which power-states could be assigned to each wait-state in order to study potential power savings. Although their work focuses on benchmarking CPU power performance, exploitation of idle states is something which every system component can take advantage of, including and especially disks in storage systems.

An energy-aware I/O interface, called CIAO, is proposed by J. M. Kundel et al. [KMKL12]. CIAO aims to provide developers a tool which makes their applications energy-aware while hiding the hardware's low-level details behind a library. The work is presented as an extension on top of the existing ADIOS [LZKS09] interface, which is already used in several scientific applications for providing I/O optimizations. The CIAO interface requires users to annotate the code to mark computation and I/O phases manually. By using the developer's information about the application, namely estimations about the durations of each phase, the framework is able to make better choices that result in greater energy savings by making better use of resources, and putting unused devices into low power states.

Btreplay is another tool which is being used in order to play back workload traces. In the paper presented by Verma et al. [VKUR10], actual power consumption is measured using Watts up PRO Power Meter and Btreplay for replaying traces. In their scenario, disks are assumed to be either in active or idle mode; with two different states with two different consumption levels, which are equal across all disks. For performing event tracing, tools such as Xperf tracks both processor and disk usage. Xperf is used in order to measure performance degradation as well [SKZ08].

2.3.3 Power estimation models

Current approaches for analyzing power usage and estimating energy consumption fall into different categories: power modeling at the hardware level [AAF+09, MMB13], power modeling at the performance counters level [LGT08], and power modeling at the simulation level [SJC+14, PGC+13, MKL10].

Simulation techniques are commonly used for evaluating both performance and energy consumption. Prada et al. [PGC+13] describe a novel methodology that aims to build fast simulation models for storage devices. The method uses as starting point a workload and produces a random variate generator that can be easily integrated into large scale simulation models. A disk energy simulator, namely Dempsey [ZSG+03], reads I/O traces and interprets both performance and power consumption of each I/O operation using the DiskSim simulator. Dempsey was only validated on

mobile disk drives. This solution predicts energy consumption using the simulated disks characteristics instead of system metrics.

Manousakis et al. [MMB13] present FDIO, a feedback-driven controller that improves DVFS for I/O intensive applications. This solution relies on the node being instrumented for obtaining fine-grained power measurement readings. Their feedback controller detects I/O phases and quickly switches the CPU frequency to all possible states, and selects the optimum setting power/performance ratio. While our work also describes the physically instrumentation to obtain fine-grained power readings, we use this instrument to analyze data movement patterns and detect power regimes. Our proposed model does not require having this kind of invasive instrumentation in order to predict energy consumption.

Lewis et al. [LGT08] uses the inherit node temperature to predict energy consumption. The authors discuss the interaction of the different components for their modeling. The authors propose using read and writes per second metric (obtained by iostat) for modeling I/O workloads. In this paper we demonstrate that the dirty buffer size metric can be used for modeling write power usage. Allalouf et al. [AAF+09] develop a scalable power modeling method that estimates the power consumption of storage workloads (STAMP). The modeling concept is based on identifying the major workload contributors to the power consumed by the disk arrays. In contrast, our solution models the energy consumed by the I/O stack, in addition to storage devices.

For estimating energy consumption, some works also focus on system performance counters [CM05, ERKR06, LJ03]. These works propose linear models that are able to provide run-time power estimations, and are validated with instrumented hardware. Other works concentrate on reducing energy consumption of individual storage devices. Zhu et al. [ZDD+04] optimize disk energy consumption by tuning cache-replacement strategies that increase idle time and reduce disk spin-ups.

Other works concentrate on large-scale storage systems, whose main goal is to achieve power proportionality. Narayanan et al [NDR08] demonstrate a power proportional storage system that is capable of absorbing a large number of writes. A large number of works have proposed models for individual components such as CPU (e.g., [CBBA10], [MB06]), hard disks (e.g., [ZSG+03]), and networks (e.g., [WPM02]). In addition, for a large body of modeling techniques at both node-level and distributed-level we refer the reader to [OAL14], [SL13], and [LBIC13]. Our work focuses on exploring, analyzing, and modeling the power consumption of the operating system's I/O stack across all components. Unlike most works mentioned in this section, we do not provide a generic power model for computation, or limit our analysis to a single system component, but focus on energy consumption caused by data movement patterns across the memory hierarchy and I/O stack. Like Li et al. [LBC+14], we aim to build models that help better understand and contribute to reducing the energy consumption of the storage stack.

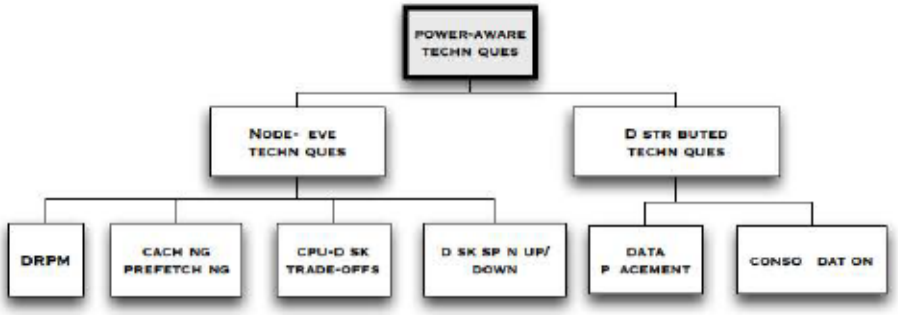


Figure 2.4: *Taxonomy of power-aware techniques.*

2.4 Taxonomy of power-aware techniques

We present a high-level taxonomy of power-aware techniques, depicted in Figure 2.4 [LBIC13]. This taxonomy primarily distinguishes two main categories: *node-level techniques* and *distributed techniques*. The reasoning behind this is that most techniques can be rigorously classified into these two categories, whereas it is much more difficult to classify a technique into either *power-proportionality* or *energy efficiency*.

Node-level techniques can be classified into caching, prefetching, I/O scheduling, and disk spin up/down. The purpose of caching and prefetching techniques is to store frequently used data in the fastest and more energy efficient memories of the storage hierarchy, thus decreasing requests to main storage. Similarly, disk scheduling techniques re-schedule I/O operations in order to prolongate disk standby times. The longer standby times produced by these techniques increase the effectiveness of disk *power-cycling*, a technique that saves disk power by spinning the disk down during idle times in order to save power. However, disk spin-down is not supported throughout all enterprise-level storage solutions due to limited driver support, as evidenced by [NDR08]. Disk power-cycling has been shown to increase the probability of disk failure, because disks can usually spin-up and spin-down only a limited number of times [PWB07, AS12, JS08]. Furthermore, spin-up/down delays can cause important data access latencies due to the substantial time the process can take, especially in real-time based applications. Even energy can be a cause of concern when spinning a disk up or down, as the reduced time a disk spends idling is not always sufficient in order to compensate the power required to spin the disk back up, resulting in lack of power savings. This is usually due to the fact that disk spin-down time is too short compared to break-even time [WYZ08, GZS+03, CPB03, GSKF03, ZZ05, LDM01]. When taking into account that each operation of transitioning to and from low-power modes (commonly known as standby mode for hard disks) has an additional power penalty, the decision to transition into such modes is non-trivial. Section 2.5 details solutions related to this issue. Dynamically adjusting disk rotation speed [GSKF03] (DRPM) is a technique which can be described as power-proportional, because it adapts the disk speed to meet performance requirements. Our work mainly focuses on node-level techniques, addressing energy efficiency in multiple layers of the oper-

ating system: CPU-level techniques, efficient virtualized data sharing mechanisms, and power prediction models.

Distributed techniques include *data placement layout strategies*, which help achieve *resource consolidation*. The purpose of consolidation is to maximize power proportionality by having a smaller number of nodes working at a higher utilization rate, instead of having many at a lower utilization rate, which is highly inefficient. However, this raises challenges related to handling I/O bottlenecks [NDR08], peak loads, availability, and elasticity. While some distributed storage systems aim for consolidation by migrating workloads [BF07, LBMN09, MCRS05, NDR08, SBP+05], this technique can become a difficult task when applications need a significant amount of state [KB10], because that state data would need to be migrated along. Data migration, fine or coarse-grained, might have an impact on network and storage systems due to the additional stress it generates. Furthermore, data placement strategies of existing distributed storage systems often prevent powering off nodes and obstruct consolidation because data is replicated and nodes are kept online to ensure availability and high I/O throughput [LK10]. Solutions which address these challenges are detailed in Section 2.6.1.

Node-level energy efficiency techniques and distributed power-proportional techniques share the common goal of generating idleness in order to power off storage subsystems. For this reason, they are usually complementary. Tsirogiannis et al. [THS10] and Harizopoulos et al. [HSMR09] define energy efficiency as the ratio of useful work done to energy used. The authors of Rabbit [ACG+10] consider a storage system to be ideally power-proportional when “the performance-to-power ratio at all performance levels is equivalent to that at the maximum performance level". Figure 2.5 shows the energy efficiency and power consumption of a system as a whole. This figure helps to illustrate the concept of power proportionality very well. The green line represents how power consumption varies with system utilization. It clearly shows how the system still consumes half of the peak power while completely idle. The energy efficiency (ratio between utilization and power) is therefore very low at that point, and increases with higher system utilization. Thus, the system shows poor power proportionality. A figure representing an ideal power proportional system would depict a power consumption level which starts at 0 when the system is idle, and a constant energy efficiency ratio for any utilization level. The same concept of power proportionality applies to distributed storage systems, in which many systems collectively become very inefficient when idle, and more energy efficient the closer the system gets to peak performance.

Often, the issue of whether a technique fits into energy efficiency or power proportionality is fuzzy, as both concepts are correlated, and a case could be made which argues for either of the two categories. This is the reason why we chose to classify power-aware techniques based on node-level and distributed system techniques.

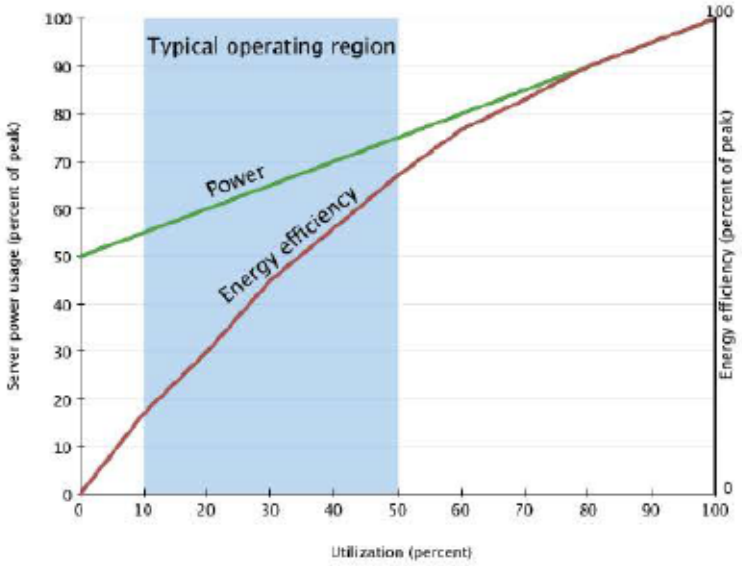


Figure 2.5: *Power usage vs. hardware utilization demonstrates how inefficiencies make power usage distance itself from the power-proportional ideal. Based on data provided by [BH07].*

2.5 Node-level storage energy efficiency techniques

While there are many solutions that attempt to make a large distributed system power-proportional, other researchers focus on making individual computer systems and components more energy efficient or power-proportional through software-based approaches. In the realm of storage systems, the energy efficiency and power proportionality of storage devices is paramount, which is why many researchers focus on hard disks. The challenge resides in the fact that disks are not power proportional [GBG+10, THS10], and that the low-power modes of disks incur a significant amount of delay as explained before. In spite of SSDs being power proportional and more energy efficient [THS10], SSDs are not expected to replace hard disks as the primary storage medium in the near future due to the fact that their cost-per-byte is up to three orders of magnitude greater [NTD+09, AGSS11]. In addition, the reduced capacity of solid state disks hinders meeting the demand of ever-growing data storage space [AGSS11]. This section therefore focuses on traditional hard disks and trade-offs between performance and energy efficiency.

The first trade-off is found when varying the number of storage devices. Applications which rely heavily on storage, such as databases, experience greater performance when increasing the amount of hard disks. However, the performance gain decreases with each additional disk, while each new disk adds a fixed amount to the power budget [HSMR09]. This results in a break-even point between energy efficiency and performance, where the most energy efficient point is below the performance peak. Figure 2.6 depicts this trade-off when varying the number of hard disks for a database management application.

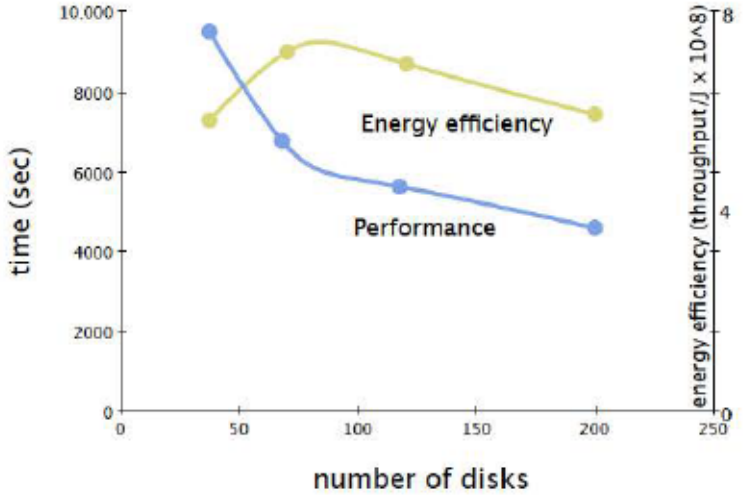


Figure 2.6: *Time and energy-efficiency vs. number of disks for TPC-H Throughput Test. Data obtained from [HSMR09].*

Even for a fixed number of hard disks, software can be fine-tuned for achieving greater energy efficiency by making use of hardware power-saving features. Similarly to CPUs, which make use of Dynamic Voltage Frequency Scaling (DVFS) [BPSB00, CC07] in order to reduce power consumption, disks usually consist of four main power modes. Many vendors include sub-states for each mode which provide additional power modes [PS04]. The main four states are called *active*, *idle*, *standby*, and *sleep*. Reads and writes occur at full performance while in the *active* state. The *idle* state keeps the disk spinning, but electronics may be partially shut down and disk heads may be unloaded. This state provides a near-instant switch to active state, although power savings are not great because the disk is still spinning. When a disk is switched to *standby* it is spinned down. While not completely off, most of the electronics and mechanical parts are stopped, thereby consuming a small amount of power. However, spinning the disk back up can take a considerable amount of time, in the order of seconds, and depends on the hardware. While a *sleep* state consumes the minimum amount of power, a hard reset is required in order to get the disk to higher states. In practice, most disks consume the same amount of power when in standby and sleep. Table 2.2 shows disk power consumption for a typical disk. While the absolute power values vary across different disk models and manufacturers, most have a standby power consumption value that is equal to that of the sleep state.

In light of this, the challenge is clear. Disks are far from being power-proportional because they are very energy inefficient when idle. However, spinning the disk down can result in undesired latencies. Mismanaging when to spin a disk up/down can even have a negative effect on power consumption, as spin-up is the most energy expensive operation. Therefore, a policy which decides when to transition from idle to standby can have a significant impact on power management. The most commonly used transition policy is the threshold policy. A timeout (or a time threshold) t is set, which determines the amount of time a disk is allowed to remain idle before

Vendor	Western Digital	Seagate	Samsung
Disk model	WD30EZRX	ST3000DM001	HD322GJ
Active state power (W)	6	8	5
Idle power (W)	5.5	5.4	4.2
Standby power (W)	0.8	0.75	0.8
Sleep power (W)	0.8	0.75	0.8
Spin-up consumption (A)	2	2	2
Spin-up time	–	–	8 s
Load/Unload cycles	300,000	300,000	50,000

Table 2.2: *While absolute power consumption values vary among three representative disks, standby state and sleep state usually have equal power consumption.*

being spun down. One popular and still widely used solution is the 2-competitive algorithm based on break-even time [FKL+91], which sets a timeout equal to the disks's break-even time. The reason for choosing the break-even time as threshold is that $T_{break-even} = \frac{E_{up}+E_{down}}{P_1}$ seconds is known to be optimal for deterministic algorithms, E_{up} and E_{down} being the amount of energy required to spin a disk up-/down, and P_1 being the power required to keep disk spinning while idle [ISSG05]. An algorithm is said to be c-competitive if its cost is no greater than c times that of an optimal off-line algorithm [FKL+91]. Another popular algorithm is the randomized algorithm [FKL+91], which randomly chooses a timeout no longer than the break-even time. The randomized algorithm has proven to have the best worst-case performance, while the 2-competitive algorithm has the best worst-case performance among fixed-threshold algorithms.

2.5.1 Disk scheduling

Scheduling I/O requests to the disk differs from other disk power-saving strategies in that it does not require a specific data placement, or data to be migrated or reconfigured. This is an advantage because power-aware I/O scheduling does not require applications to adapt, and works transparently.

For example, Chou et al. [CKR11] complement the solutions that attempt to save energy by spinning disks down to standby mode, with power-aware scheduling strategies that determine disk location for incoming requests of the storage system. Their work consists of a scheduling strategy that aims to maximize expected disk standby time and minimize energy consumption, without interfering with existing data placement and other power saving techniques. Their approach takes a stream of requests as input, and determines the disk location for serving each request, trying to minimize energy consumption. The scheduling problem, which is NP-complete, is formally defined and solved using efficient polynomial approximations. Three different strategies are discussed: online, offline, and batch processing variants. The latter two are shown to be solvable through weighted independent-set and weighted set-cover algorithms respectively. The batch and offline strategies include a cost function which tunes the existing trade-off between performance and energy con-

sumption. Their results show that this trade-off ranges from achieving a reduction of energy consumption of 35% on one end, to halving response times on the other end. The technique also results in a reduced number of spin-up/down operations.

Another technique which aims to reduce disk energy consumption is DRPM, which consists in dynamically modulating the disk spin speed [GSKF03]. While most techniques consider transitions from a spinning idle disk to a stopped disk in standby state, DRPM adjusts disk rotation speed to match the required performance. This is estimated by taking into account the number of queued requests periodically. Gurumurthi et al. provide a detailed analysis of disk energy consumption and how disk rotation speeds affects power consumption [GSKF03]. While the term *power-proportionality* was not widely used before 2007, it is clear that this approach increases the power-proportionality of hard disks. Unfortunately, this technique is implemented within the disk controller, and dynamic speeds are often not available to the operating system, which limits its applicability [SK06]. Furthermore, due to physical and cost constraints, DRPM drives are not being commercialized in quantity [GBG+10].

At the RAID level, it is possible to re-schedule I/O requests in order to take advantage of data redundancy spread along the disk array in a way that will maximize disk standby times. Conventional power-saving techniques miss the overall picture of redundant data, while EERAID [LW04] employs caching and RAID-awareness to schedule requests to disk which are operating in high-power modes, giving the remaining disks a better chance of remaining in low-power mode. For instance, in a RAID 1 setting, the authors propose a novel power-aware read dispatch policy in which N requests are dispatched to each disk alternatively. The goal is to alternatively have one active disk taking in read requests, while the other disks can stay in low power states. Every N requests, roles are swapped. In a RAID 5 setting, EERAID is primarily focused on the destaging process, during which cached update operations are flushed to disk. In this case, blocks that are to be written to disks in a high-powered state are preferably destaged and evicted from cache. Blocks whose disks remain in a low-power state are given preference to remain in the cache, therefore prolonging the time disks are allowed to remain in a low-power state.

2.5.2 Power-aware caching and prefetching

Strategies which focus on storing data in faster, volatile memories can help increase the energy efficiency of storage media as well. Most of these strategies can be classified into caching or prefetching. Caching is useful because it helps to reduce disk utilization as a whole, while prefetching data into memory usually helps to increase disk standby times. By generating an extended period of idle time and preventing small time spans of inactivity, an implementation can more easily switch disks from an idle state to standby, where it is spun down. Spinning down a disk requires the idle period to be sufficiently large because spin-up and spin-down time can cause significant latencies (in the order of a few seconds), and can even cause energy inefficiencies as well, since accelerating disk plates takes significantly more power than keeping them turned off or even spinning. Consequently, what all strategies have in

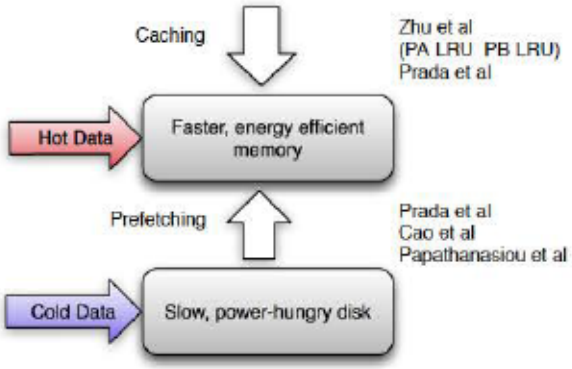


Figure 2.7: *Caching and prefetching Strategies aim to increase energy efficiency by decreasing hard disk use.*

this section have in common is that they will try to gather *hot* (frequently used) data in lower, energetically cheap memory, while usually leaving *cold* (infrequently used) data in hard disks, as depicted in Figure 2.7.

In the context of caches, a replacement policy has to determine which data block to replace when the cache memory is at full capacity. LRU (Least Recently Used) is a widely used policy due to its combined simplicity and effectiveness. There are power-aware cache management algorithms which extend LRU, focusing on power consumption. For example, Zhu et al. [ZZ05] propose two algorithms, PA-LRU and PB-LRU. Their main idea is to take into account workload characteristics that affect energy cost, such as cold misses and arrival time distribution. As a result, their variants are 22% more energy efficient than LRU as a cache replacement algorithm, while offering better response times for online transaction processing (OLTP) workloads as well [ZSZ04, ZZ05]. The main difference between PA-LRU and PB-LRU is that PA-LRU estimates the energetic cost of replacing blocks, and tries to generate miss sequences which minimize energy usage [ZDD+04]. PB-LRU [ZSZ04] focuses on dynamically estimating an energetically adequate cache partition size for each disk. PB-LRU performs just as well as PA-LRU and requires significantly less parameter tuning, being therefore easier to adapt for existing storage systems.

Prefetching strategies which favor energy efficiency do not necessarily penalize performance, as evidenced by the work exposed in [PS04]. Based on the ideas proposed by Cao et al. [CFKL95], the work of Papathanasiou et al. [PS04] suggests a set of rules which make prefetching strategies energy-aware without compromising performance. One of the rules determines that prefetch operations should be carried out if blocks are available for replacement. This prevents the disk from spinning down in situations where the standby period would not have been properly exploited for being too short. An additional rule ensures that prefetch operations will never interrupt spun-down disk unless delaying the operation degrades performance. Consequently, the new set of rules decrease power usage by generating longer periods of disk standby time, while maintaining the hit ratio and performance.

Other power-aware solutions exist which rely on multi-tiered caches and data

storage [WYZ08]. The goal is to have the higher tiers, which contain all of the data, in a power-saving mode for extended periods of time, and have a lower tier, which is more energy efficient, serving the majority of data access requests. In addition, systems which rely on parity-based redundancy are able to exploit this, recovering data blocks from parity data stored in an active storage components. As a result, accessing disks in a low-power mode is avoided, which would have resulted in a high latency request and switching one or more disks to a higher power-consuming state.

Multi-tiered solutions which focus on caching and prefetching are based on hybrid HDD and SSD devices, using the latter as a non-volatile, faster, smaller capacity caching storage media. The advantage of this particular design is that it allows the caching and prefetching strategies to focus on increasing SSD usage and decreasing HDD usage. This results in energy and performance benefits due to the fact that an SSD drive is faster, has lower latencies, and is power-proportional, while increasing HDD standby times. For example, Prada et al. [PGGC11] use the SSD as a block cache, absorbing writes in a log-based manner, therefore avoiding erase operations, which results in less invalid pages in blocks and less overhead. The prefetching module is in charge of reading a number of data blocks using a single I/O operation. The purpose is being able to satisfy a fair number of read requests without cache misses during a relatively long period of time, during which the magnetic disk will be able to remain idle or even in standby mode.

Most of the techniques described in this section are complementary. Caching and prefetching solutions usually have no negative impact on performance. The only trade-off is in cost, particularly SSD-based solutions, and implementation effort for algorithmic-based solutions.

2.5.3 CPU-based optimizations and trade-offs

Several methods exist for trading lower storage space requirements and network traffic for higher CPU usage through compression. This usually translates into a space vs. time trade-off. One of those techniques is data de-duplication, which consists of eliminating redundant data, replacing those chunks with pointers to the original and only copy of the data [CAKLR11]. While compression strategies achieve lower storage footprint and can improve storage systems by reducing I/O pressure and network traffic, their implications on energy footprint are not trivial to determine due to the fact that these strategies are creating a trade-off between I/O overhead and CPU overhead. L. Costa et al. [CAKLR11] compare a procedure which will scan a database with and without compression. Empirical results show that while the compressed version offers greater throughput of data, the slower, uncompressed version is more energy efficient. This is due to the fact that different system components have quite different power consumption levels. A CPU-bound operation, such as processing compressed data, will consume more energy than a disk-bound operation.

In addition, the similarity ratio of data plays a big role. Compressing a file with low similarity ratio will have a negative impact on energy efficiency because the resulting file size will not compensate for the CPU overhead. For a certain similar-

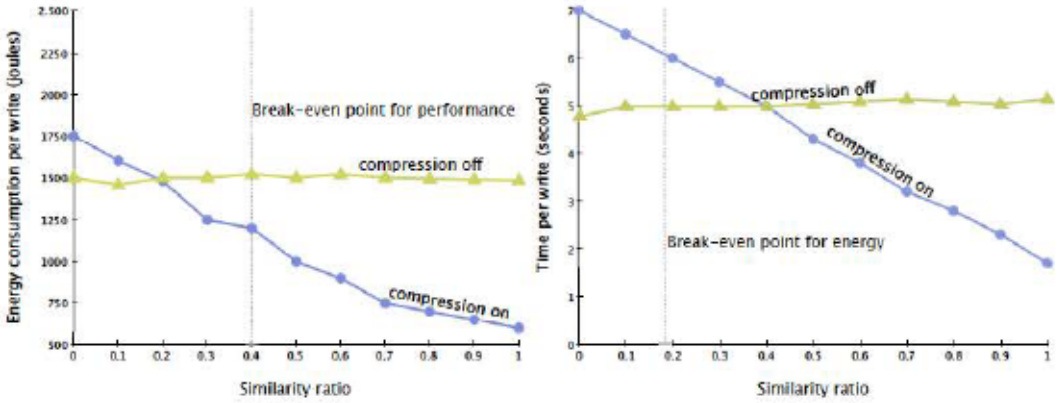


Figure 2.8: *Different break-even points for performance and energy [CAKLR11]. The graph on the left depicts energy efficiency, break-even being at about 0.2 similarity ratio. The graph on the right depicts performance with a break-even point at 0.4 similarity ratio. The data similarity ratio affects the performance of the data de-duplication algorithm, with a higher ratio resulting in a smaller file write, and hence less I/O.*

ity ratio, the smaller file write will break even with the CPU overhead. However, the break-even point for performance and for energy efficiency is not the same, as evidenced by [CAKLR11]. An example of such behavior can be seen in Figure 2.8, where each figure depicts, for the same experiment, different break-even points for energy and performance respectively. Therefore, it becomes apparent that one must take different design decisions when architecting a system, as well as when designing an algorithm. Using compression will only be beneficial for data with a relatively high similarity ratio, although the break-even point for energy efficiency is easier to reach than the break-even point for performance.

2.6 Distributed storage energy efficiency techniques

This section discusses techniques which are used to reduce energy consumption in distributed storage systems. While this thesis focuses on node-level solutions, our future goal is to work on energy and power-aware distributed solutions. The techniques detailed in this section differ from node-level techniques in the following way. Instead of optimizing energy consumption by focusing on a single component or a single node, distributed techniques optimize power cluster-wide by collectively managing the power state of all nodes and disks. The remainder of this section is divided in three parts. Section 2.6.1 discusses solutions which target power proportionality in file and storage systems either by extending existing data placement strategies or by designing novel data layouts. Section 2.6.2 presents file systems which have been explicitly designed for energy efficiency. Finally, Section 2.6.3 discusses power-aware data archival solutions.

2.6.1 Power proportional large-scale storage systems

As explained before, power proportionality is a desired property that can be achieved in different ways. The most obvious approach is to develop more energy efficient hardware that will consume an amount of power that is proportional to its resource utilization. However, today's hardware is still far from this goal [FWB07, BCH13, BH07]. The combination of several non power-proportional systems can be used to collectively produce a power-proportional system [TWM+08]. This is why many researchers are aiming to achieve power proportionality in large-scale storage systems through means such as power-aware algorithms and optimizations, some of which consist of a high-level system-wide solution [TDN11, VKUR10, ACG+10, KB10, KMA+11, NDR08, SKZ08, HSM+10]. The question of whether the notion of power proportionality extends to the realm of storage systems has been addressed by Guerra et al. [GBG+10] by studying resource utilization in enterprise-level storage systems under production workloads. Their findings reveal that systems spend approximately 20% of their time completely idle, and that the time spent on higher utilization levels was found to be decreasing with the level of utilization. The storage systems spent only about 2% of the time at a utilization level over 80%. This entails that there is a strong opportunity for distributed storage systems to improve energy consumption by becoming more power proportional. Most of the techniques which address power-proportional large-scale systems take the approach of studying novel data placement strategies. Power-proportional data placement, as argued next, is the core of a power-proportional distributed storage system, and introduces many additional challenges that are detailed and discussed in this section.

Traditionally, the goal in designing a data placement strategy was to improve performance, availability, and resilience. For example, HDFS allows users to specify a block replication factor, and ensures that replicas of data blocks are placed in different nodes and racks. However, traditional data layout strategies often encumber the task of achieving power-proportionality [LK10], mainly due to the fact that any node could be participating in an I/O operation, which complicates the selection process of which node or component to power down. In light of this, many researchers have experimented with different data layout strategies and designs that meet the needs of power proportionality. We begin by describing why, from our point of view, the data placement strategy is the core of a power-proportional large-scale storage system. A data placement strategy is strictly bound to power proportionality and a power on/off strategy, as it will:

- allow to power down devices without affecting availability.
- enable the storage system to sustain an acceptable performance level.
- allow fine or coarse-grained activation of devices to meet the demand curve.
- handle storage device or node failures gracefully.
- the more fine-grained a storage system can gear up/down, as allowed by the data placement. and power on/off strategy, the more power proportional it can be.

	Sierra	SRCMap	Hadoop (energy invariant)	Rabbit	GreenHDFS
Replica management	fixed	variable	fixed	fixed	fixed
Space overhead	$r-1$	$r*0.25$	$r-1$	$r-1$	$r-1$
Write availability	✓	✓	x	✓	x
Reliability	✓	✓	x	x	✓
Data classification	x	✓	x	x	✓
Heterogeneity	x	✓	x	x	x

Table 2.3: *Comparative summary which shows how all the solutions which are investigated and compared in this section differ in their design.*

All the techniques described in this section share the common goal of making a distributed, often large-scale storage system, power-proportional. It is also true for all of them, that power-proportionality is achieved by reducing the number of active nodes, while powering the rest off. Many challenges arise in doing so while accounting for the features of traditional data placement strategies, such as high throughput, availability, redundancy, and scalability. This leads to different design properties, and Table 2.3 offers a comparative overview of all the techniques which are investigated and compared in this section. *Space overhead* refers to the extra space required to store all replicated data. *Write availability* indicates whether writes always succeed during low power modes. *Reliability* indicates that the solution addresses disk power-cycling. *Data classification* indicates that popular data is not replicated equally to unpopular data. *Heterogeneity* indicates that the energy efficiency of the device is being taken into account when a candidate is being chosen for power off. The remaining part of this section details the differences among techniques, as summarized in Table 2.3, and goes into more depth to discuss even more aspects in which these techniques take different approaches to address the challenges of achieving power proportionality.

Replica management

Replica management is common to all techniques as well, because maintaining multiple copies distributed of data blocks is the only way to prevent data loss and to help achieve availability. Most solutions offer a fixed but configurable replication factor r, which is typically $r >= 3$. For example, HDFS-based solutions [KB10, LK10, ACG+10, SKRC10] have a default replication factor of 3, while in Rabbit [ACG+10] it is set to 4. The advantage to this approach is that replica management is relatively simple, the disadvantage being a large space overhead of $r-1$. There is not only a trade-off between replication factor and space overhead, but with power consumption as well, since more capacity entails providing power for more disks. SRCMap [VKUR10] proposes a more complicated and sophisticated *replica placement model* which replicates only the working data set. The number of replicas for each data set is variable according to the associated cost and benefit (number of misses, average load, and the power efficiency of the storage device). In this case, the space overhead is the amount of free space available in the storage system, which is assumed to be about 25%.

Power on/off strategy

The *power on/off strategy* consists of an algorithm which is responsible for selecting an active data set in a certain time window. The active data set will usually reside on a subset of nodes which will be powered on, while the rest of the systems might be transitioned to a power-saving state or even powered off. In order to achieve power-proportionality, the transition from/to a low power state needs to have a granularity that is capable of meeting the I/O demand curve.

Gear leveling is a power on/off strategy which is inspired by how automobiles conserve fuel by gear-shifting to upper gear levels [LBIC13]. Similarly, gear leveling takes advantage of replicated data sets, establishing a minimum gear level where at least one replica exists in a subset of disks or nodes, allowing the rest of the system to be powered down. When shifting into an upper gear level, additional resources are reclaimed and switched to an active state, making additional replicas and aggregate bandwidth available to meet the demand curve. At the highest gear level, all resources are powered up, offering the highest level of parallelism, aggregate bandwidth, and performance, but also working at peak power. PARAID [LBIC13], Rabbit [ACG+10], and Sierra [TDN11] make use of gear-leveling.

Similar to gear-leveling, SRCMap [VKUR10] replicates data sets to the free space of different storage volumes, and establishes a minimum power setting for which the availability of at least one replica is guaranteed. However, as described earlier, SRCMap only replicates the working set, resulting in a much smaller space overhead. A virtual to physical mapper takes care of tracking the location of replicated data blocks and routing access to those which reside in active nodes.

GreenHDFS [KB10] analyzes data file requests in order to classify data into a *Hot Zone* and a *Cold Zone*. The hot zone represents the working set, and is consolidated into activated nodes, while the cold zone stores infrequently accessed data and can be powered down. It is worth noting, however, that no mechanism allows for both sets of data to adapt dynamically as data popularity changes, as the separation being fixed from the start. This results in a much less flexible power on/off solution than gear leveling.

This inflexibility is also present in the work presented by Leverich et al. [LK10], in which they introduce a power-proportional HDFS. HDFS defines a set of rules, also called invariants, that deal with data placement in order to ensure high availability and performance. No single node can host two replicas of the same data-block, and at least two replicas must be located in different racks. The authors of [LK10] describe an additional invariant in order to provide an energy-aware storage system. The notion of a *Covering Subset* is introduced as a power on/off strategy. The Covering Subset represents the minimum amount of storage nodes required to satisfy immediate availability of any data object contained in that subset, even if all other nodes are powered off. While this results in substantial energy savings under small workloads, this power on/off strategy is the equivalent of a gear-leveled solution with just two gears, meaning there are no fine-grained increments of power and

Solution	Gear up/down granularity
Rabbit	fine-grained (node by node), coarse grained (number of gears)
SRCMap	fine-grained (node by node)
Sierra	coarse-grained (number of replicas)
PARAID	coarse-grained (number of disks - 1)
GreenHDFS	low (2 gears)
HDFS (Energy invariant)	low (2 gears)

Table 2.4: *Comparison of the granularity of each power on/off strategy.*

performance levels. One could therefore argue that solutions such as GreenHDFS and the Covering Subset are less power-proportional than other gear-leveled solutions [ACG+10, LBIC13, TDN11], or the approach taken by SRCMap [VKUR10]. Table 2.4 offers a comparison of the granularity employed by each technique for turning devices on and off.

Power-proportional granularity vs. fault tolerance

Power-proportional granularity vs. fault tolerance is a matter which is investigated in Rabbit [ACG+10]. The trade-off for a fine-grained activation of resources (usually powering on node by node) is reduced fault tolerance. In the event of a node or device failure, data will become unavailable. While durability will still be met, because a number of replicas exist, data availability can only be guaranteed if a small redundancy in active nodes is offered. However, this is incompatible with an ideal power proportionality, in which only the strictly necessary resources are active. Therefore, there is a trade-off between fault tolerance and ideal power proportionality which is yet to be resolved. In light of this trade-off, many solutions opt for a coarse-grained gear-leveled approach, where the ideal power proportionality is sacrificed in order to ensure data availability in the event of hardware failures.

Write availability

Write availability can become an issue for power-proportional storage systems due to the fact that writes intended for those nodes that are powered down will result in a write miss. All solutions which address this issue make use of *Write off-loading*. D. Narayanan and A. Donnelly [NDR08] originally proposed this technique when observing that the storage layer is dominated by write requests due to the effect of memory caches that are able to absorb the majority of read requests. For write operations to succeed, and in order to maintain an appropriate write I/O bandwidth, most nodes would need to remain powered on, therefore preventing power saving techniques from being successful. *Write off-loading* addresses this problem by temporarily redirecting (i.e. off-loading) writes to different storage volumes. The benefit of using write off-loading is twofold. First, it allows writes destined for de-activated storage systems to succeed. And second, it removes potential write bottlenecks by being able to off-load writes to any active server. In addition, disk spin-up penalties

for write requests are masked out due to the nature of the off-loading solution. The distributed log to which writes are committed allows writes destined to offline nodes to succeed, allowing those nodes to remain powered down. Off-loaded data blocks can be lazily reclaimed when the nodes are brought back online, or synced in the background during low activity periods. Sierra [TDN11], SRCMap [VKUR10], and Rabbit [ACG+10] resort to write-offloading for solving this problem. Other power-proportional data placement strategies which do not employ write off-loading but address data availability and write bottlenecks suggest on-demand power up of nodes [LK10]. In our opinion this is an inferior solution due to the expected latencies, which puts into question whether this solution addresses availability at all. Furthermore, no alternative solution for mitigating the major power spike that would result from a single node failure has been proposed so far.

Reliability

Reliability is another issue pertaining to hard disk drives. Spinning up disks too often could adversely affect their reliability [PWB07, AS12, JS08]. This is an important design aspect because hard disk usually have an expected maximum number of start and stop cycles. Coarse-grained power cycling strategies help reduce the number of daily spin-ups, which ensures disks reach the expected lifetime. Because all techniques ultimately end up powering disk drives on and off, this is a problem which needs to be addressed by all solutions. This is explicitly addressed by Sierra [TDN11], SRCMap [VKUR10], and GreenHDFS [KB10], all of which ensure that the power up/down intervals are long enough.

Data classification

Data classification is a process of categorization which can be used to differentiate data. In Table 2.3, it indicates whether data are being handled equally. A power-proportional storage system can use data classification to its advantage by employing popularity-based replication, as SRCMap does [VKUR10]. This has the benefit of reducing replication space overhead, reducing system capacity requirements and avoiding unnecessary work and power consumption when replicating data sets. However, data classification entails additional system complexity and possible data processing overheads, which have not been detailed in depth by any of these works. GreenHDFS [KB10] relies heavily on simple data access metrics for classifying the data cluster into hot zone and cold zone, helping achieve consolidation and yielding great power savings.

Heterogeneity

Heterogeneity is another property which can be taken advantage of, as different hardware often offers different energy efficiency and power proportionality. Because the whole point of a power proportional storage system is being able to have only

Problem	Solution
Fine-grained proportionality	Gear leveling, Virtual disk mapping (SRCMap)
Avoiding bottlenecks in the event of replica failure	Data placement that allows reasonable rebuild parallelism
Hard disk reliability	Avoid spin-ups through coarse-grained consolidation intervals, Write off-loading, Caching
Write availability	Write off-loading
Storage space replica overhead	Less replicas of data not belonging to working set
Fault tolerance and power proportionality	Power up in incremental groups e.g. Gear leveling

Table 2.5: *Solutions which have been proposed to common problems are highlighted in this table.*

a subset of machines working, leaving inefficient hardware powered off is preferable. In spite of this, almost no technique employs this strategy, with the exception of SRCMap [VKUR10]. However, for this strategy to be effective one would need to manually input energy efficiency and power proportionality parameters for each range of components, as hardware devices do not offer a way for software to query these data.

Overview of power-proportional data placement problems and solutions

Finally, an overview of problems and their corresponding solutions is offered in Table 2.5, which summarizes all the issues and proposals which have been detailed in this section. Similarly, Table 2.6 provides an interesting map, which relates which techniques and optimizations are better suited for different types of workloads as far as power proportionality is concerned. This table includes techniques described in other sections, which are complementary to the ones described in this section. Techniques such as write off-loading or opportunistic spin-down can incur delays (for example, due to disk spin-up times), which means that they are not suitable for workloads which have short default timeouts and are sensitive to peak response times. Solutions that employ techniques such as consolidation or migration might not be suitable for workloads which are unstable (i.e. they can vary unexpectedly) due to their coarse-grained operation. Workloads that require good storage performance might not fare well with techniques such as adaptive disk speeds and data de-duplication, which can produce overheads which lengthen response times.

2.6.2 Energy-efficient file systems

File systems are used for organizing data in data storage devices. Distributed file systems are widely employed in large scale computing systems. In the context of high performance computing, parallel file systems are widely used. While all of them have been designed in order to maximize performance, few of them have been designed in order to target the problem of energy efficiency. Some relevant examples are revised

Sensitivity to Avg. Resp. Time	Sensitivity to Peak Resp. Time	Stability of Workload	Techniques					
			C	T	S	W	A	D
Yes	Yes	No						
		Yes	x	x				
	No	No			x	x		
		Yes	x	x	x	x		
No	No	No			x	x	x	x
		Yes	x	x	x	x	x	x

Table 2.6: *Techniques:* **C***:* Consolidation*,* **T***:* Tiering/Migration*,* **S***:* Opportunistic Spin-down/MAID*,* **W***:* Write Off-loading*,* **A***:* Adaptive Disk Speeds*,* **D***:* De-duplication*. Data obtained from [GBG+10].*

next.

While energy efficiency is not the primary goal, GreenFS [JS08] is able to save up to a 60% of the peak power related to storage. In order to accomplish this, disks have to remain powered down in standby mode as much as possible. GreenFS reads and writes are preferably served over the network, at the GreenFS server. This is due to the fact that network transfers under a hundred megabytes are more power efficient than disk transfers. Small network transfers being the majority of I/O operations, GreenFS achieves disk power savings for the clients. However, the system relies on a remote parallel file system storage, from which all clients pull data from, and whose power consumption is not described in detail. Clients improve performance and improve storage energy efficiency by reading and writing to and from local flash storage. The flash layer acts as a local cache and is the preferred layer to read and write from due to its high energy efficiency and power proportionality. The disk is spun-up only periodically several times a day in order to synchronize changes and maintain local persistency, and remains in standby the rest of the time. The hard disk is only used in the event of network failure or when network performance is not sufficient. The roles of the backup server and local storage are inverted in GreenFS, the local disk is mostly used for data recovery scenarios, while the remote server is used whenever local cached data in the flash layer is missing. GreenFS achieves overall storage power savings of 60% [JS08].

Blue File System's (BlueFS) goal is to provide ubiquitous access to shared and personal data [NF04]. This is similar to other distributed file systems such as Coda [SKK+90] with a number of differences. Another goal of BlueFS is to be power efficient. The system optimizes data access performance when possible by making use of meta-information about its configured storage devices in order to maximize both performance and power savings. The logic of how to gather requested data and where to retrieve it from is performed by a user-space process called Wolverine. For deciding which device to access, both performance and power are taken into account. A weighted sum of access time estimation and energy cost is taken, resulting in a read from the device which offers the best performance and lowest energy cost, according to the weighted cost function. In order to determine the energy usage of a particular operation, BlueFS resorts to an offline device characterization strategy

which provides estimates for each device operation. While less precise than online power measurement strategies, power estimation is far more simple and only requires each device to be characterized once, if ever, due to device specifications usually being of great help in providing good enough estimates for selecting the right device.

Lastly, Ge [Ge10] evaluates the energy efficiency of network and parallel file systems. They observe that application I/O access characteristics and data layout are very influential factors for the energy efficiency of a file system. In light of this finding, one can exploit knowledge of application I/O characteristics in order to specify the data distribution and layout strategy of storage devices.

2.6.3 Energy efficient data archival

As the world's data grows exponentially [GR11], there is an inevitable need for long-term data archival. For businesses, data preservation is often mandated by law, and cloud services which store user media file such as pictures must keep up with a growing need for efficient and reliable long-term storage. While traditional backups archives were designed for *write-once, read-rarely*, newer archival solutions need to address *write-once, read-maybe* workloads [SGMV08]. Tape provides poor random-access performance, which is why modern solutions resort to using hard disk drives.

Due to the fact that a large number of hard disk will consume a great deal of energy [LGLZ12], solutions exist which arrange data in such a way that infrequently used data reside in disks which are powered off, while hot data will reside in a small subset of drives. The active data set can either consist of cached data, such as in MAID [CG02], or migrated data, such as in PDC [PB14]. Caching has the benefit of not requiring data migration, which can degrade energy savings for long migration intervals, but determining the appropriate number of caching disks is more difficult. Hibernator [ZCT+05] leverages multi-speed drives and focuses on meeting performance goals by determining optimal disk speeds and even spinning disks up when necessary. In contrast, Pergamum [SGMV08] focuses much more on energy efficiency and reliability, and less on performance. Their approach for improving energy efficiency is to add flash-based NVRAM storage to every node for persistent, low-latency metadata storage. In addition, they rely heavily on low-power CPUs, which manage data redundancy and integrity checking in a distributed fashion. That way, storage controllers and power-hungry high-class CPUs are dispensed with. Power savings for these techniques are as high as 85% when compared to traditional RAID storage clusters [CG02]. Consequently, these techniques might not be good enough for workloads which expect a high number of Input/Output operations per second (IOPS), due to the increased delay incurred by disk activation, which is why they are mainly used for data archival. In the context of most energy-efficient data archival techniques, performance, while important, is secondary to energy efficiency. However, each solution is fine tuned for a different performance-efficiency trade-off. Table 2.7 offers a summarizing overview of the main approach to achieve energy efficiency used by the data archival solutions discussed in this section.

Solution	Approach to energy efficiency
Hibernator	multi-speed disks
MAID	caching
PDC	data migration
Pergamum	NVRAM, decentralized controllers

Table 2.7: *This table highlights the different paths taken by each technique in order to save energy.*

2.7 Storage I/O virtualization

This section describes common solutions for storage virtualization in use today in environments running virtual machines. In these environments, it is common that a virtualization software layer called *hypervisor* multiplexes host physical resources among several virtual machines running *guest* operating systems. Common storage I/O virtualization solutions can be classified in two categories: block-level virtualization (labeled as A1, A2, and A3) and filesystem-level virtualization (labeled as B1, B2, B3). Each of these categories is detailed in the following sections.

2.7.1 Block-level data access

This section details different block-level storage virtualization solutions depicted in Figure 2.9, from left to right.

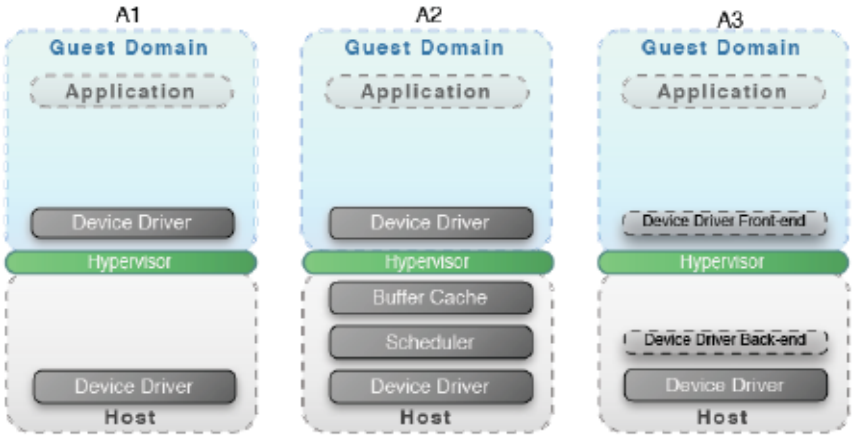


Figure 2.9: *Block-level storage virtualization solutions in use today.*

A block-level device can be fully virtualized either on top of a physical device driver (A1) or on top of a file system (A2). These solutions emulate a device driver in the host and have the advantage of flexibility (a virtual disk can be mapped to any physical device), but suffer performance penalties due to duplicated functionality. Most virtualization platforms support emulated devices for backwards compatibility, while offering paravirtualized device drivers for high performance [WR12]. Paravirtualized device drivers (A3) eliminate the need for duplicated device drivers. In most modern virtualization platforms their implementation is split into a front-end run-

ning in a guest and a back-end serving front-end requests and running the device drivers natively. Paravirtualized drivers can be used for solutions A1, A2, B1, and B2, in order to reduce device emulation overhead in the guest. Block-level virtualization solutions are detailed next.

(A1) Logical disk in the guest domain is mapped to a physical disk

This is usually a very efficient configuration for achieving near native performance, as the hypervisor assigns a device to a single virtual machine. Consequently, I/O can be directly mapped to the guest's memory. However, this solution does not allow sharing a partition, complicates efficient data sharing between guest domains, and has the drawback of physical devices being more difficult to scale than logical devices.

(A2) A logical disk in the guest domain is mapped to a file on the host file system

This solution has the benefit of being able to easily manage snapshots, as the virtual file system is just a file residing on the host file system, decoupling logical disks from physical resources. However, it is not feasible to share the file system across different domains other than read-only. Inefficiencies are present due to the fact that data blocks are written through the host file system. A file system performs a lot of redundant operations, while the application only needs block-level access. Additional memory copies, translating paths to inodes, block allocation, file metadata, and reading/writing inodes are considered unnecessary overhead. The device drivers are duplicated as well.

(A3) Paravirtualized device drivers

Paravirtualized device drivers remove the need for duplicated device drivers. Device drivers in the guest domain are replaced by a shim layer (a device driver front-end) which communicates with the host (running a device driver back-end), runs the device drivers natively, and multiplexes access of all paravirtualized guest domains. However, these improvements do not help domains share data efficiently or control the data path past their domain, once it reaches the host domain. Paravirtualized drivers can be used for solutions A1, A2, B1, and B2, even when the target device is not a physical disk, in order to reduce device emulation overhead in the guest. Paravirtualized device drivers are available for most virtualization solutions, such as KVM virtio drivers[Rus08], Xen PV split drivers [BDF+03], and VMWare's guest tools. Parallax [MAC+08] is a related block-level virtualization solution that provides sharing of virtual disk images (VDI). In Parallax, a device driver backend can be a remote host. Parallax does not support write-sharing of VDIs. In turn, it offers efficient operations for fine-grained frequent snapshotting of VDIs.

2.7.2 Filesystem-level data access

This section details different filesystem-level storage virtualization solutions depicted in Figure 2.10, from left to right.

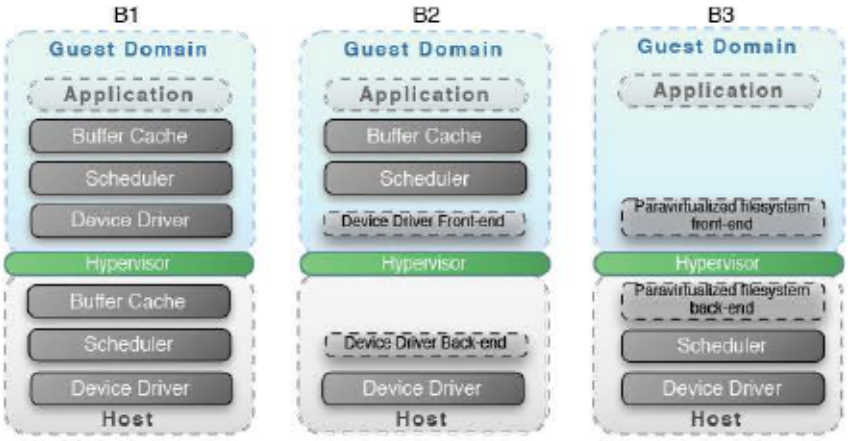


Figure 2.10: *Filesystem-level storage virtualization solutions in use today.*

(B1) A file system in the guest domain is stored directly on top of a disk partition of the host

This solution has the same drawbacks as A1, but offers filesystem-level access to the upper layers.

(B2) A file system in the guest domain is stored in a file on the host file system

This solution is known as nested file systems, and much has been written about its inefficiencies (up to 67% performance degradation) and the difficulties of fine tuning for performance [LHW12]. Storing a file system inside a file of another file system entails the following complications which result in performance penalties: redundant I/O operations, duplicated and uncoordinated disk schedulers, redundant memory copies, duplicated buffer caches. An additional drawback is that applications running on the guest domain do not have any control over the data flow through all storage layers. For example, during our experimentation with the Xen hypervisor we observed that it was not possible to ensure that data was synced to disk when using Xen's *file://* driver. Write benchmarks which use *fsync()* system calls to ensure that data is actually being written to disk would report a throughput which was an order of magnitude greater than what the disk is physically capable of. This is due to the lack of coordination between virtualized and host I/O stacks for this particular driver, where *fsync()* semantics are not being honored outside of the virtual environment.

(B3) Paravirtualized file system

In a *paravirtualized file system* [PGR06, JVHLP10], the guest's file system is aware of the virtualized environment and the host works in coordination with all guest domains. In such an environment, data can be efficiently transferred across all guest and host domains. Filesystem-level coordination between domains means that data can be exported or shared efficiently, effectively eliminating redundant memory copies, memory allocations, and duplicated storage layers. For example, using a paravirtualized user-level data access mechanism, an application can read data from disk into a memory buffer which can be zero-copied to another domain. Data can be written to disk specifying a write policy, such as write-through or write-back, without incurring additional memory copies. The proposed work of this thesis fits in this category. We propose a set of novel techniques, I/O semantics, abstractions and mechanisms that enable coordinated and efficient sharing of data across domains. With our solution and methodology we aim to improve the programmability, performance and energy efficiency of intra-domain I/O and data sharing. We demonstrate this with various collective I/O operations in Chapter 5.

2.8 Memory sharing mechanisms in virtualized environments

In virtualized systems, virtual machines often need to communicate data to and from the host, and share data among each other. This is especially the case in paravirtualized systems, which use paravirtualized drivers that reduce the virtualization overhead of emulating hardware and software components. Instead, virtualization solutions often opt for a split-driver approach, as depicted in Figure 2.11.

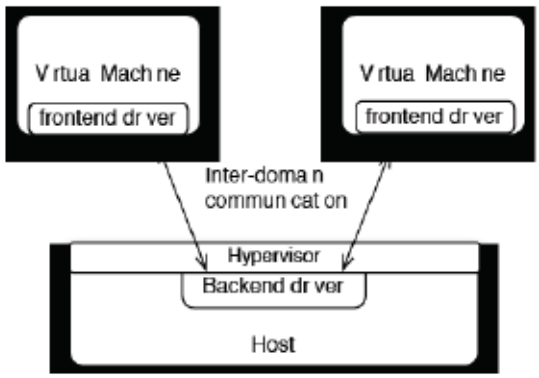


Figure 2.11: *Split-driver approach for paravirtualized drivers.*

KVM [Rus08] and Xen [BDF+03] use this approach for their paravirtualized solutions. While any application running in a virtual machine can use a TCP/IP socket to communicate with another virtual machine, the networking stack incurs a large overhead when compared to traditional UNIX sockets. Of course, UNIX sockets do not provide inter-domain communication (i.e. they can not communicate across virtual machines or domains), thus the need for an inter-domain communication

channel. The inter-domain communicator is in most cases implemented with shared memory. However, conventional shared memory mechanisms may not be used due to the fact that virtual machines are designed to be isolated from each other. Therefore, host and hypervisor setup machine frames which are shared between two virtual machines. While initially this was achieved by the use of a technique called *page-flipping*, in which the page ownership is transferred from one domain to another, this mechanism was found to incur a high overhead [MST+05]. Instead, pages are shared in a way that two domains can access the same memory frame directly. The same memory frame is then accessed through different page numbers in each of the virtual machine's virtual address space. In Xen, this is achieved by using grant tables [SKKJ07]. This shared memory is then used to implement circular ring buffers in order to support high-throughput VM to VM communication with low latency and overhead, which is ideal for paravirtualized drivers.

This mechanism has also been used in order to implement solutions that provide inter-domain communication with lower overhead [LJSL09, HJL+08, DPNJ09, KKJ+08, HKGP07, ZMRG07]. For instance, XenSocket [ZMRG07] is a solution which uses shared memory ring buffers to provide a socket implementation which is binary compatible with existing applications and outperforms TCP/IP sockets. XWAY [KKJ+08] presents a very similar approach. While both solutions reduce performance overhead, two memory copies are at least required to transfer data from one domain to another: One data copy operation to the ring buffer (sender write), and another from the ring buffer to the target domain (recipient read). The approach described by Youseff et al. [YZW08] is different from the previously described approaches because the shared memory buffers are exposed to user-space applications, instead of hiding them behind a POSIX interface. Hence, this solution sacrifices isolation and protection to gain performance. The proposed method also incurs less hypervisor calls, which has the benefit of simplifying the API and improving performance by reducing copy operations from and to the shared memory buffers. However, the lack of POSIX compatibility breaks Application Binary Interface (ABI) and Application Programming Interface (API) compatibility, and the interface does not take the opportunity to introduce new data sharing semantics.

In this thesis, we also propose a novel memory sharing mechanism that allows data to be shared by more than two domains at a time. A shared memory address space is created where data can be zero-copied and accessed by any number of domains using novel abstractions and semantics.

Chapter 3

Analysis and modeling of energy consumption in the I/O stack

Energy efficiency in the context of computing has recently become a hot topic for academia and industry. Systems ranging from portable devices to large-scale distributed systems, are all constrained to operate within a tight energy budget. In the case of large-scale computing systems, the International Exascale Software Project has studied the challenges that need to be addressed for reaching *exascale* [D+11]. One of the main barriers is the non-computational aspect of energy, which is mostly related to data movement, not computation. It is also known as the *energy barrier*. Because today's facilities operate on a 20 MW power budget and it would be difficult to provide greater power supply, that fact has been set as a design target [KBB+08]. Consequently, systems need to reach an energy efficiency of $50\,GFlop/Watt$. As of 2015, the most energy efficient supercomputer is still off by a factor of over 7 [Gre15b, Hem10].

While many aspects need to be improved in the context of energy efficiency, including low-power electronics, memory, CPU, applications, cooling, and the data center itself, data movement was found to be one of the biggest energy-related challenges. In addition, storage systems are expected to play an increasingly important role because the world's digital data grows exponentially, doubling every year, as revealed by a recent study conducted by the IDC [GR11]. This results in a special interest in optimizing storage systems in data centers, and motivates the need for more research aimed at improving the energy efficiency of storage technologies. In spite of this, there is no sufficient research that studies the energetic impact of Input/Output (I/O) of data across the I/O stack.

While our long-term goal is to analyze the power consumption related to data movement in large distributed systems, we argue that it is first necessary to understand how energy is consumed within a single node. Motivated by this fact, this

chapter focuses on analyzing power consumption across the entire storage data path. In this chapter, we explore how energy is consumed in the storage data path by vertically analyzing power consumption across the whole I/O stack, and not just the plugged storage devices. Numerous other works have already studied energy consumption of system components such as storage devices, DRAM, and CPUs, and power consumption models for HPC applications have been presented in the past [OAL14]. Our work focuses on exploring, analyzing, and modeling the power consumption of the storage data path in a single node across all system components when performing I/O in order to study the energetic impact of data movement in a detailed manner. We demonstrate how we are able to apply data exploration and analysis techniques in order to identify distinct power regimes during I/O operations. We use this knowledge to build models that accurately predicts energy consumption for different I/O operations and access patterns. This work also discusses the methodologies that have been followed in order to build the models, detailing the physical instrumentation that produces fine-grained power measurements that are used for analysis, as well as the software instrumentation that captures performance data metrics in a non-intrusive manner. Our hardware instrumentation is capable of reading power consumption derived from the whole motherboard, including main system components such as memory, CPU, and hard disks.

The remainder of this chapter is structured as follows. Section 3.1 offers a global overview of the work provided in this chapter. Section 3.2 details how power and performance data were gathered, and gives insights into the data collection methodology that was followed. Section 3.3 focuses on analyzing how power is used by processes and the operating system under different I/O patterns. Section 3.4 details a methodology for automatically extracting out of all gathered data the most relevant metrics for I/O and data movement, showing correlations between system metrics and power usage. Section 3.5 presents a power model based on these observations, which accurately predicts the power used when doing I/O operations. Section 3.6 evaluates the proposed model using different storage devices and I/O access patterns. Finally, Section 3.8 summarizes our contributions and presents future work.

3.1 Overview

This section gives a global overview of the goals, scope, and methodologies provided in this work. Our main motivation is that energy consumption in the software I/O stack is not well understood. This is in part due to the vast amount of variables that affect energy consumption. Modeling I/O power consumption is difficult due to the many inputs or control knobs that play a role. Anything from the I/O access pattern, operating system's internal algorithms, to the underlying hardware impacts the resulting power consumption. Therefore, we start by analyzing the data gathered from simple micro benchmarks in order to *characterize*, *understand*, and *model* energy consumption. We approach this problem with the following methodology.

First: Data are collected by running a series of micro benchmarks on the instru-

mented system. The data are checked for coherency and consistency; if necessary, this step is repeated to obtain corrected and coherent data. This corresponds to Section 3.2.

Second: We perform an exploratory data analysis to explore and understand power consumption across all system components when performing various I/O access patterns. We identify distinct power and performance regimes, and provide a detailed explanation for the existence and transition between these regimes. This corresponds to Section 3.3.

Third: We propose a methodology for automatically extracting relevant information from all collected data, and apply this methodology to identify the most relevant system metrics out of all collected data for each access pattern. The goal of this process is to test the understanding of our previous analysis, and expand it with actual data. This corresponds to Section 3.4.

Fourth: The knowledge extracted from the previous steps are used to build I/O power models. We start by developing simple analytical models that can be combined for mixed workload applications. For writes, a simple approach is often inaccurate due to the complicated nature of how data is moved across the I/O hierarchy, so we provide a second, more sophisticated model. This corresponds to Section 3.5.

Fifth: Our models are evaluated for its accuracy and prediction capabilities using various micro benchmarks, and we provide a detailed evaluation that compares the effectiveness of the two proposed write models. We also show how the energy consumption of applications making use of mixed I/O patterns can still be predicted using a combination of our models. This corresponds to Section 3.6.

Sixth: Because multi-threaded I/O workloads show very different behavior than single-threaded I/O workloads, we take a different approach for modeling these applications. We demonstrate and evaluate how the energy consumption of several multi-threaded FileBench macro-benchmarks can be predicted using our proposed model. This corresponds to Section 3.7.

3.2 Instrumentation and data collection

This section details the process of gathering power usage and performance data, the experiments that were run while obtaining these data, and how the different data is explored analytically. The system under test (SUT) is instrumented in two different ways in order to gather necessary data: 1) hardware instrumentation, to collect power usage data passively, and 2) software instrumentation, to collect various performance data with low overhead.

3.2.1 Hardware instrumentation

We collect power usage data from two hardware instruments. First, a power measurement board is placed between the power supply unit (PSU) and the motherboard.

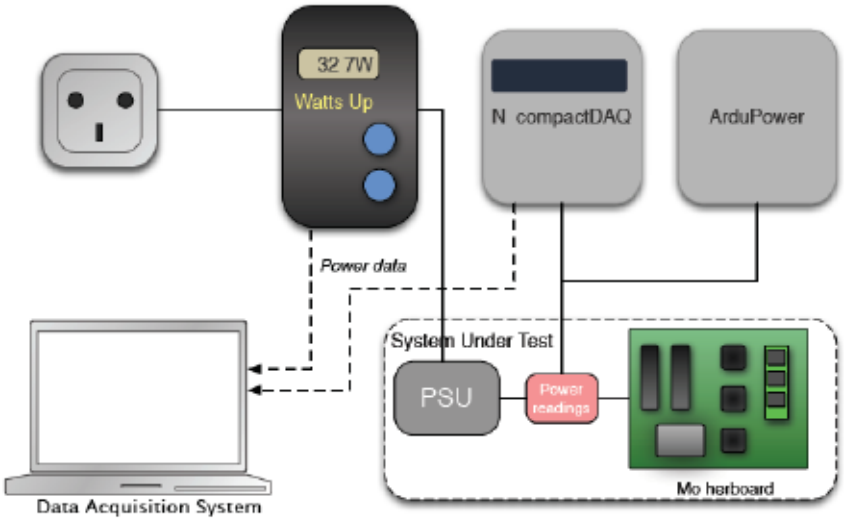


Figure 3.1: *Power measurement instrumentation setup.*

This instrument measures currents and voltages, and feeds the data through a National Instruments cDAQ-9174 data acquisition device to an external system over a USB interface. This setup is inspired by PowerPack [GFS+10] and is depicted in Figure 3.1. The power measurement board contains hall-effect current sensing components that are known to suffer from offset errors. Therefore, this error was measured using known voltage and current sources using a voltage reference device, and corrected automatically in software while calculating power. While the hardware is able to generate samples at up to 15 Khz, we found 1 Khz to be sufficient precision and more manageable for processing. Second, a Watts Up Pro unit is connected to the PSU. This is an external power meter that provides readings of the aggregate power consumption, including PSU, at 1 Hz. Because the external device only provides samples at 1 Hz, we focused on the more detailed internal readings and used the external meter for validating the readings obtained by the internal device.

3.2.2 Hardware platforms

We performed the analysis, developed our model, and tested our methodologies on a system running Linux 3.16.1, a 4-core Intel Ivy Bridge processor, and 8 GB of DDR3 memory. During analysis, modeling and testing we used different storage devices with different performance and power characteristics. We focused on three different storage devices; two hard disk drives from different generations and manufactured by different vendors and one SSD. Specifically, these were a Western Digital Caviar Blue 500 GB SATA III 7200 RPM with a 16 MB Cache (HDD1), a 2 TB TOSHIBA MK1002TS with a 64 MB Cache (HDD2), and a Kingston 120 GB SSDNow V300 (SSD).

We also used a second system where we were able to reproduce our results, predictions, and methodology. A Nehalem server platform equipped with 2× Intel Xeon

X5560 (a total of 8 physical cores) clocked at 2.80 GHz, 12 GB of DRAM memory, and a Seagate Barracuda 500 GB SATA II HDD. This system not only has a different micro architecture, but also a different power meter and power measurement framework. The power meter, named ArduPower [DHKF15] is an Arduino-based low-cost internal wattmeter similar to our own setup that functions as both power measurement board and data acquisition device. It offers fine-grained (480 to 5,880 Hz) power measurement by leveraging Allegro ACS713 hall-effect sensors, and providing 16 channels to monitor power usage of system components. To obtain power usage data from this power meter, we leverage the PMLib framework, a well-established package for investigating power usage in HPC applications [BBC+13].

3.2.3 Software instrumentation

Software instrumentation consists in gathering performance data and hardware state information provided by the operating system in a non-intrusive manner. In this study, we collected time series of data from a variety of operating system interfaces, which we call system metrics. Data stemming from *procfs*, *sysfs*, and *cpufreq* drivers are read periodically. These interfaces in the Linux kernel are exposed to user space in the form of read-only files. These files contain by convention ASCII encoded structured data that are updated by the operating system every 100 ms. These data include detailed CPU usage information such as c-state (CPU sleep states according to ACPI standard) and p-state (CPU performance states in ACPI standard) usage time [JS03], virtual memory information (size of cached data, dirty buffers, free memory, write back buffers), time spent in user processes and system time, page faults, number of context switches, number of processes running, and number of interrupts. Collecting this information results in a time series, which details system state and performance over time. The overhead of gathering data at 10 Hz was about 1%. This overhead is calculated experimentally.

In addition, we gather data using the Linux perf framework, sampling data at 5Hz. We determined empirically that for our experiments, 5Hz provided a good balance between time resolution and the number of performance counters to be collected. This is because the perf framework will only report counter values that have reached a certain threshold. Perf values provide complementary information to the data gathered from *procfs* and *sysfs*: hardware performance counters (such as instructions, cache references, cache misses, cycles, branches, etc) in addition to soft performance counters providing information from several kernel subsystems, such as the CPU scheduler, the block layer, and interrupts.

3.2.4 Data collection, exploration, and cleaning

Experimentation starts with running micro benchmarks in order to stress specific components of the system. The goal is to determine which power lines draw more current when using CPU, DRAM, and disks over SATA. Being able to associate certain power lines with a specific component is very useful during an exploratory data analysis. This helps better understand the per-component power usage when explor-

ing power usage plots, yielding a better comprehension of power usage across the I/O stack. These micro benchmarks are followed by storage I/O micro benchmarks which stress the I/O storage layer using different workloads. These are detailed in Section 3.3.

A text file with structured information is kept which contains information about each experiment. For each experiment, data file paths for power and performance data are referenced alongside metadata such as running time, I/O pattern, data size involved in the I/O workload, average read/write throughput, name of the benchmark and test being run, command line parameters, etc. This document is then read by R scripts, where experiment data and metadata can be loaded together in a structured way before being processed.

Due to the fact that data are being gathered from several sources, relating these data may first require synchronizing the time series. Synchronizing the time series consists in making sure that the data are aligned in time, and putting everything into the same time scale. Time synchronization is addressed in two ways. First, our system metric gathering framework orchestrates data collection from different sources within the system. Since all data are collected using the same clock, the framework can ensure that the data are synchronized. Second, two machines gathering data are synchronized over Network Time Protocol (NTP), ensuring that their clocks do not drift.

When relating data of different time scales, there are two approaches. The first consists in reducing the longer time series by taking averages to match the time scale of the shorter time series. The second consists in interpolating the data of the shorter time series to match the scale of the longer time series. For our intents and purposes, we found both approaches equivalent. Under both approaches, the computation that relates time series of different time scales offered equivalent results.

After the data are gathered, effort is put into a process called *data cleaning*. Data cleaning is necessary before performing data analysis in order to detect missing or incorrect information. These inconsistencies are handled either by repeating the experiments to obtain all the necessary data correctly, or by taking missing data into account during analysis. In our case, only the former was necessary. Careful data cleaning revealed software instrumentation bugs in several occasions, which we were able to detect by checking the data for coherency and consistency. These were detected, for instance, by cross-checking distinct but related data provided by the operating system, such as CPU usage with p-state and c-state information. The exploratory analysis of the data may also reveal "impossible" values (such as negative values where those do not make sense, or values that report disk write speeds above the hardware's rated performance). Usually, as this process involves a bit of data exploration, the need for additional performance metrics in order to complement or cross-validate the data will become apparent. This results in an iterative process consisting of the following steps: 1. Developing software instrumentation 2. Running experiments to gather new data 3. Exploring and checking data for consistency.

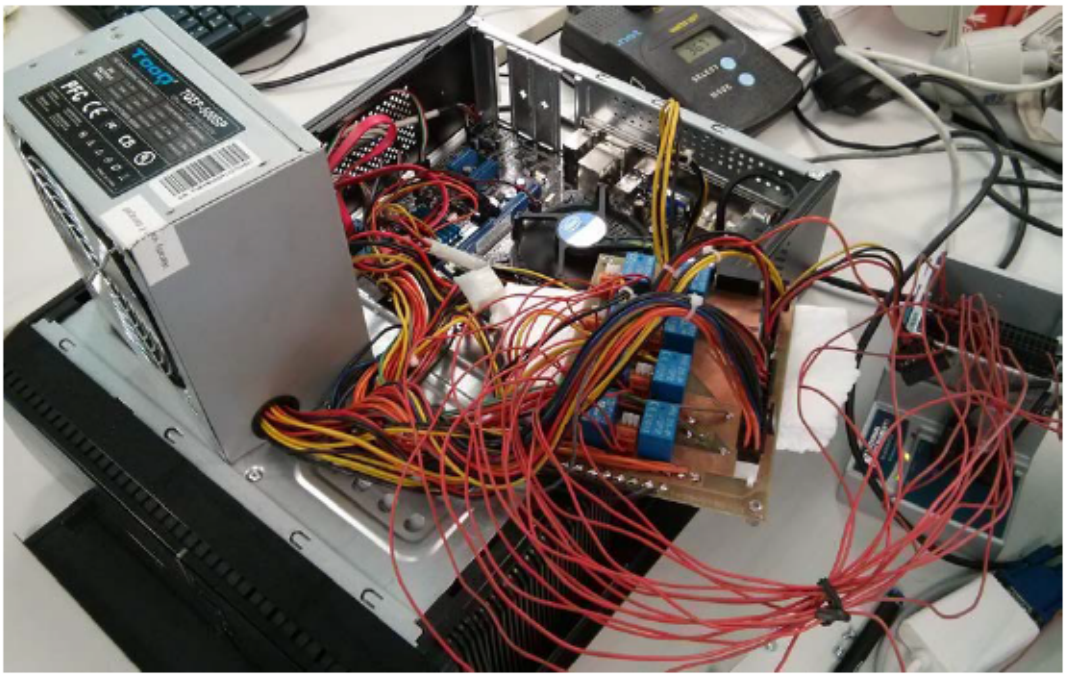


Figure 3.2: *Photo of our power measurement setup.*

3.3 Data Exploration and Analysis

Through exploratory data analysis we acquire knowledge about how the data are managed by the operating system and how this affects power usage and performance. More precisely, our goal is to explore the data to understand how the electrical power is consumed in a computing system to perform I/O operations. This process should help us understand the behavior of power consumption within I/O operations, and how power consumption varies across different I/O access patterns. In order to measure data movement, we run simple I/O micro benchmarks while collecting data as detailed in section 3.2. To carry out these benchmarks, we leverage `fio` [Axb], which is a commonly used micro benchmarking tool developed by the lead developer and maintainer of the Linux block I/O subsystem.

In order to stress the I/O storage layer in different manners, we run the following micro benchmarks: 1) read a file sequentially using different file sizes. 2) write a file sequentially using different file sizes. 3) read a file randomly using different file sizes, and 4) write a file randomly using different file sizes. For each experiment, we also vary the amount of data already present in the page cache from 0% to 100%. When performing I/O, we always use the POSIX interface. File sizes range from megabytes to tens of gigabytes, and the size of I/O operations varied from 4KB to hundreds of megabytes. However, we found that varying the size of I/O operations did not have a measurable impact on power usage. This is expected, as the operating system will split and merge block I/O requests before they reach the device driver.

Initially, we compare and explore the power usage of different system compo-

nents such as disks, CPU, and DRAM. Examination reveals that disk power is only dependent on the type of I/O workload being run, and is independent of the power usage of other components. When comparing HDDs to SSDs, we were able to detect the same power regimes, with the unsurprising observation that the SSD performed better and consumed less power. We also observe that DRAM power usage is strongly correlated with CPU power usage. We found that CPU plots are great for exploration because power usage will vary greatly in a very short time frame, producing fine-grained details in plots, whereas DRAM plots are similar, but much noisier. This is not surprising, considering that CPUs are known to be the most power proportional hardware component.

Initial exploration of detailed CPU data suggests that for each I/O pattern (corresponding to the aforementioned micro benchmarks), the system is driven into a different power and performance regime. Due to the different I/O paths taken by read and write I/Os, we examine and analyze them separately.

3.3.1 Read workloads

Exploration of a very power-proportional component such as the CPU enables us to distinguish even short-lived power and performance regimes, such as the CPU plot shown in Figure 3.3, where we perform strided reads from disk and memory, alternatively. The transition between power and performance regimes (corresponding to disk and main memory access) are clearly visible. This experiment can be replicated by performing a sequential read of a 4GB file, where 128MB every 128MB were page cache hits and data were not moved from disk. This results in strided access to both disk and memory in 128MB blocks.

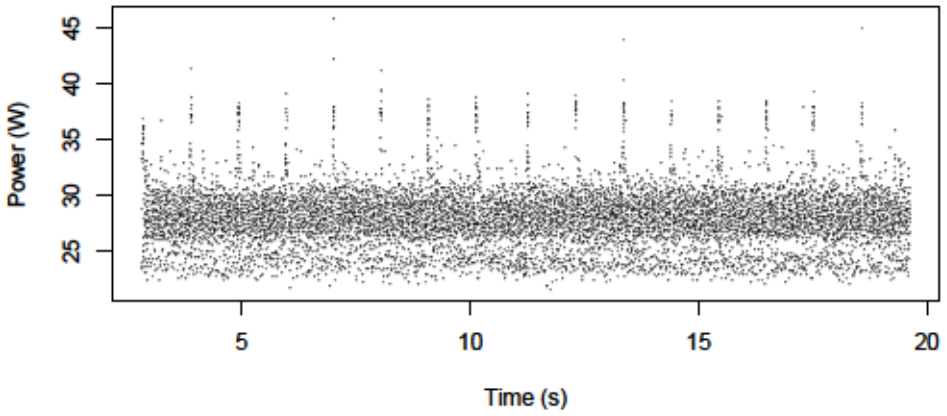


Figure 3.3: *CPU power during read of a 4GB file. Read access pattern of 128MB blocks and a 256MB stride accesses memory and disk, alternatively.*

Hence, we observe two different power and performance regimes for each component: when taking the I/O path to main memory in order to reach data residing in the page cache, and when reaching data stored in disk. Varying the amount of data present in the page cache changes the amount of time spent in each regime

proportionately.

3.3.2 Write workloads

Power plots exploration reveals that for write I/O patterns, the system switches between distinct power regimes over time. Write I/O operations exhibit interesting irregular patterns due to the way the operating system manages data while it moves across the I/O stack. Figure 3.4 depicts power regimes for a sequential 4GB file write. This is an important difference from read workloads, which are more easily categorized into distinct regimes, and will remain on the same regime unless the data is using a different I/O path (such as the previous example with the strided disk and memory read). Instead, writes drive the system to transition between multiple power and performance regimes, even when there is no cached data in main memory. Due to this difference in the way data is transferred through the memory hierarchy by the operating system, we find that write operations are more difficult to analyze and model.

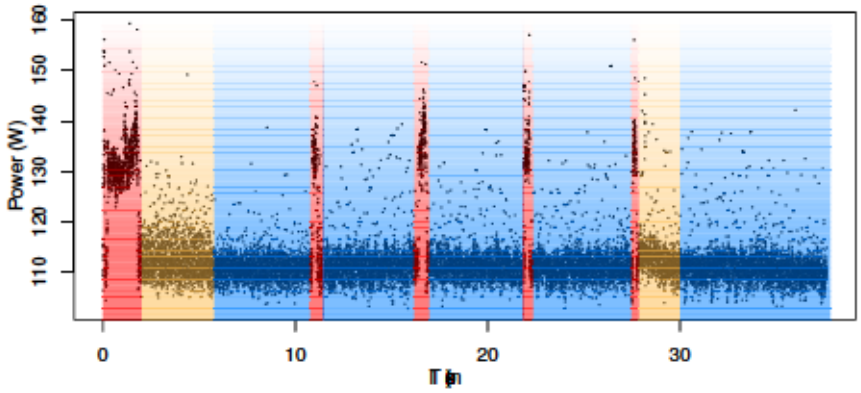


Figure 3.4: *Power regimes during a sequential write of a 4 GiB file.*

We identify different power and performance regimes that correspond to temporal regions in these write I/O operations. Even for simple write I/O operations, power consumption can vary significantly over time. This is due to the fact that the system transitions between different power and performance regimes while data moves from main memory and is written to disk. These regimes show that a simple straw-man approach to modeling writes using average power and write duration as input would not be sufficient, especially for short write operations. A comparison of a simple straw-man approach with a more sophisticated model we propose is detailed in Section 3.6. This also motivates the need to have a way for estimating the power consumption of I/O operations.

In order to provide more context into the process of transitioning between regimes during write I/O operations, we detail the operating system's algorithms that govern how data is moved in the following section. We hypothesize that for write workloads, dirty memory management is a key aspect to power consumption, and will confirm this hypothesis in the analysis and evaluation later on.

Writes and dirty memory management

The operating system's goal is to maximize write performance and yet limit the total amount of dirty memory. Therefore, the operating system's algorithms will throttle write operations according to the state of dirty memory, disk bandwidth, and dirtied pages per second [Wu14]. These dirty memory throttling mechanisms are directly related to the power and performance regimes described in Section 3.3.2, and are detailed next. The throttling algorithm uses two thresholds for the amount of dirty memory (*freerun* and *limit*), resulting in three distinct regions, as depicted in Figure 3.5. Each region exhibits different behavior for system write operations.

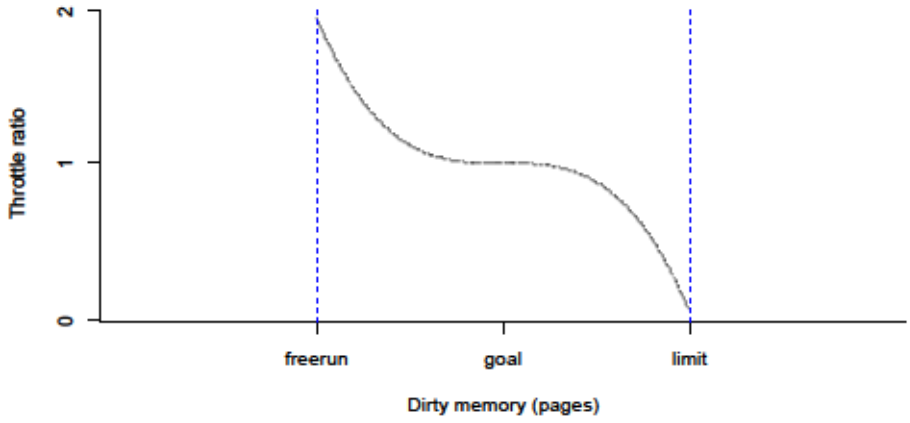


Figure 3.5: *Analytical representation of the Linux dirty memory rate limiting and write throttling function. When a memory percentage less than* `freerun` *is dirty, no throttling occurs. Between* `freerun` *and* `limit`*, writes are throttled with the goal of matching the disk write bandwidth. After* `limit` *is reached, writes are blocked.*

These regions are defined as a function of two operating system variables, *dirty_background_ratio* and *dirty_background*. in Figure 3.5, *limit* equals *dirty_background*, *freerun* is defined in Equation 3.1, and goal is defined in Equation 3.2.

$$freerun = \frac{dirty\_background\_ratio + dirty\_background}{2} \tag{3.1}$$

$$goal = \frac{freerun + limit}{2} \tag{3.2}$$

While in the left region of Figure 3.5, a process will be able to dirty memory, unthrottled, until the threshold *freerun* is reached. Whenever the total amount of dirty memory is above the *dirty_background_ratio* threshold, the operating system will start to write back dirty data in the background.

The middle region of Figure 3.5 throttles writes according to a cubic polynomial,

shown in Equation 3.3.

$$f(dirty) = 1 + (\frac{goal - dirty}{limit - goal + 1})^3 \tag{3.3}$$

This function has the goal of throttling writes more heavily when more memory is being dirtied, and less heavily when less memory is being dirtied, while trying to settle in between at the goal, which matches the disk write-out bandwidth. *dirty_background* equals *limit* and controls when writes are hard throttled and blocked (right region of Figure 3.5).

Finally, when in the third region, the amount of dirty memory is above *limit*, and all processes block when issuing write operations until a lower threshold is reached by the background write back kernel processes.

Discussion of write I/O power and performance regimes

The operating system's throttling mechanism results in different power consumption and performance regimes within write operations. This explains the power regimes that are visible during a 4GB file write in Figure 3.4, as shown during our analysis. We identify 3 distinct power regimes (other than idle). The regime with the highest power usage is colored in red. During this time, data is being buffered in main memory. As we have pointed out, there is a strong correlation between an increase of dirty buffer size in the page cache, and power spikes when data is being written to the page cache. We will detail this observation that was made during data exploration with data in Section 3.4. In the second regime, colored in yellow, we observe two different power states that look like two parallel bands. During this period, the operating system is performing I/O in the same manner as in the second regime. In addition, it keeps adding data to the page cache, as in the first regime, albeit at a slower pace, which explains a higher power usage than for the synchronous I/O of the second regime. In the third regime, colored in blue, no data are written to main memory. At this point, the process blocks until sufficient dirty data has been flushed to disk, while the operating system performs I/O in the background.

3.4 System metrics analysis and selection methodology

In this section, we propose a methodology for analyzing I/O operations and automatically obtain the most relevant system metrics for power consumption automatically. The goal is to validate our understanding of the previous analysis, and expand it based on actual data. System runtime data gathered through software instrumentation amount to 120 system metrics. Every system metric represents one aspect of the system as a discrete time series for the duration of each experiment.

Using the data collected as a time series, we design a methodology in order to detect metrics that are highly correlated with power consumption, following a similar approach as in [DKCC15]. Identifying these metrics is important for developing new power usage models that are more sophisticated than the aforementioned straw-man

approach. This methodology leverages the Pearson's correlation and consists of the following steps:

1.- For each collected system metric, we calculate its correlation with power consumption.

2.- For each collected system metric, we compute the derivative w.r.t. time and calculate its correlation with power consumption.

3.- Every correlation whose absolute value is less than an empirically determined threshold t is discarded.

4.- The union of both correlations results in a table of system metrics that are relevant for power usage during data movement of I/O operations.

Note that in the design of the methodology, we have taken several aspects into account. First, some metrics are cumulative values, and therefore monotonically increasing with time, e.g, number of interruptions occurred since boot. Others are defined as a rate or instantaneous value and might vary with time accordingly, e.g., power consumption varies depending on the load of the machine. Therefore, a methodology should be aware of this fact in order to avoid correlations of metrics of different nature, and convert cumulative time series to instantaneous values so all metrics can be compared. The transformation from accumulated to instantaneous values is normally performed by subtracting the previous observation r_{t-1} from the current one r_t at the time t.

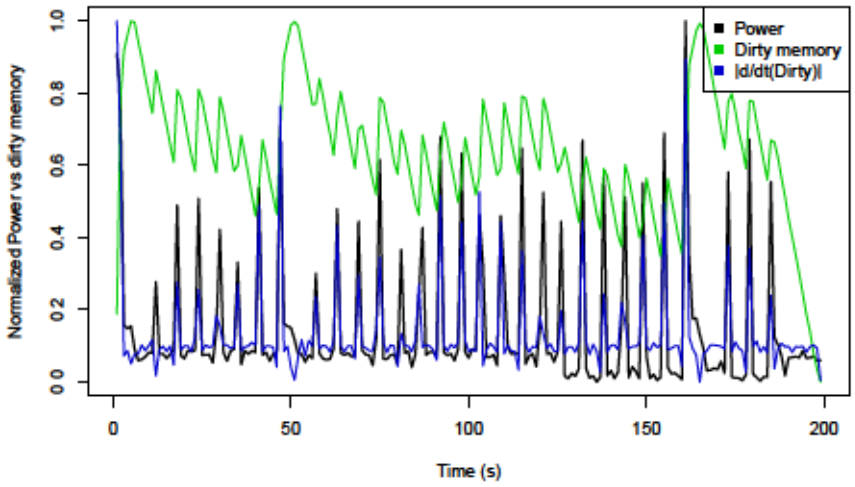


Figure 3.6: *Dirty memory, $|d/dt(\text{dirty memory})|$ and power consumption of a sequential write of a 20 GiB file.*

Second, since data movement has a direct impact on the power consumption, metrics that are measured in quantities of data should be transformed into the rates of movement by computing their corresponding derivatives. For example, as shown in Figure 3.6, the dirty memory metric measures the number of bytes in memory that

must be written back to the disk at a given time[1]. However, the page dirtying rate (or speed) needs to be derived from the dirty memory metric in order to be compared with the power consumption properly. By calculating the dirtying rate, it is possible to measure the amount of data that any user-space application is writing to main memory. Information closely related to data movement can be obtained from the available system metrics. We believe this is a fundamental step when developing a methodology that identifies most correlated metrics with regard to the power usage.

Next, we apply our methodology and present a detailed analysis on read and write operations.

3.4.1 Analysis of write operations

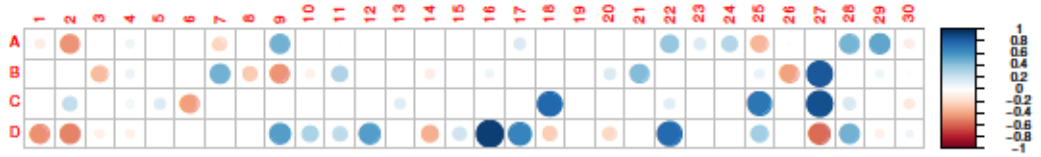
Applying our proposed methodology, we obtain one power usage time series and 120 system metric time series for every benchmark run. Figure 3.7 depicts the correlations of all the 120 system metrics with power consumption during a sequential write of a 4 GiB file. While the top plot shows the direct correlations, the bottom plot takes the derivative of the data before computing the correlation with power. Indeed, the bottom plot clearly shows that only one system metric is highly correlated with power (B4) and the rest have a very low correlation. Table 3.1 lists the most significant system metrics. As it is shown in the table, those metrics with values below our empirically obtained threshold of 0.75 have been discarded.

Table 3.1: *Correlation of system metrics to power for a sequential write.*

Metric	Corr(data)	Corr(d/dt(data))	Corrplot Tile
`cpu_system`	**0.94**	-0.20	D16
`Dirty`	0.08	**0.92**	B4
`softirq`	**0.87**	-0.12	C27
`procs_running`	**0.84**	-0.17	B27
`cpu_user`	**0.77**	-0.12	D22

In Table 3.1, `cpu_system` is the system CPU utilization, `Dirty` is the number of dirty memory pages, `softirq` is the number of Linux software IRQs, `procs_running` is the number of running processes, and `cpu_user` is the user mode CPU utilization. Not surprisingly, CPU utilization is highly correlated with power since the CPU is the most power intensive component during these operations. Interrupts are also highly correlated with the I/O power usage. However, we argue that the most relevant system metric for write operations is the derivative of the number of dirty pages. While the system CPU utilization shows a slightly higher correlation, we can hardly use this metric to reflect the level of power consumption used by I/O operations. The system CPU utilization correlates with the power consumed by other workloads running on the machine and, in consequence, this metric is not useful for decoupling the power derived from I/O from that of other computations. Similarly, the number of running processes cannot be considered a good metric due to its nature. However

[1]Note that the absolute value of the derivative is computed in order to superimpose positive and negative rates on a single normalized plot.



(a) *Raw system metrics.*

(b) *Derivative system metrics w.r.t. time*

Figure 3.7: *Correlation plot between power usage and system metrics for a 4 GiB write.*

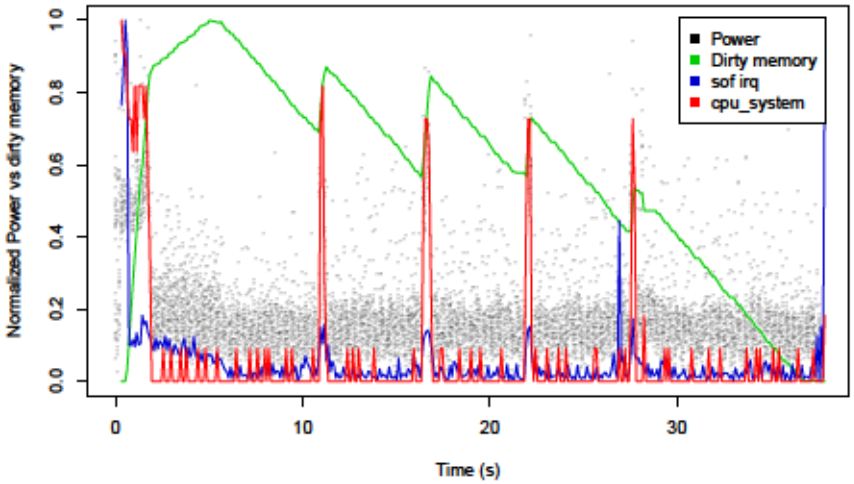


Figure 3.8: *Power consumption vs. correlated metrics for the sequential write of a 4 GiB file.*

the dirtying page rate, $d/dt(\texttt{Dirty})$, is a very useful metric for estimating I/O power consumption because it is specific to I/O and shows the amount of data that is being moved by the software I/O stack. Therefore, we conclude that the dirtying page rate is the best system metric to reflect the I/O-related power usage. Figure 3.8 shows clearly how well the `Dirty` system metric describes data movement with regard to power consumption while Figure 3.9 visually compares the other system metrics that are also highly correlated with the power time series. Therefore, we conclude that by monitoring the `Dirty` system metric, it is possible to infer the power regime of the system during write operations.

Since our wattmeter consists of several channels, it is possible to measure several power lines, simultaneously, and measure per-component power consumption as well. We complemented the study with a per-line analysis which highlights correlations at the component level. Figure ?? shows correlations of a relevant subset of system

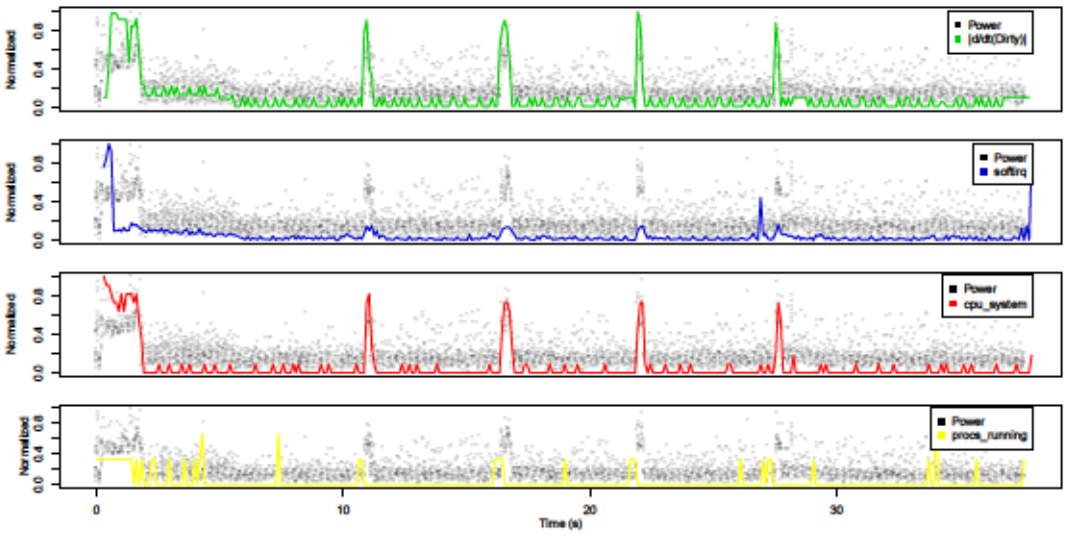


Figure 3.9: *Power consumption, dirty memory, software interrupts, CPU utilization and processes running for the sequential write of a 4 GiB file.*

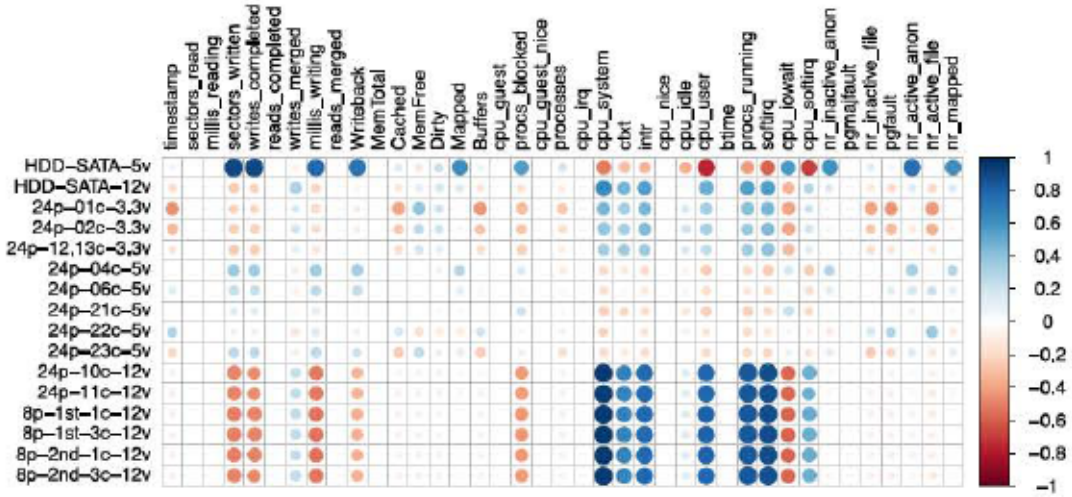
metrics for raw data (top) and derivative data w.r.t. time (bottom). Unsurprisingly, these plots are consistent with the previous correlations plots that use the aggregated power consumption. For instance, the bottom correlation plot clearly shows that d/dt(`Dirty`) is correlated with the write activity.

These plots also show which parts of the system are physically designed to provide static and dynamic power, as some of the power line rows (left axis) do not show correlations with any system metric. We are also able to verify that the data collected from these power lines show very little variability over time when the system is stressed and switches between various power and performance regimes.

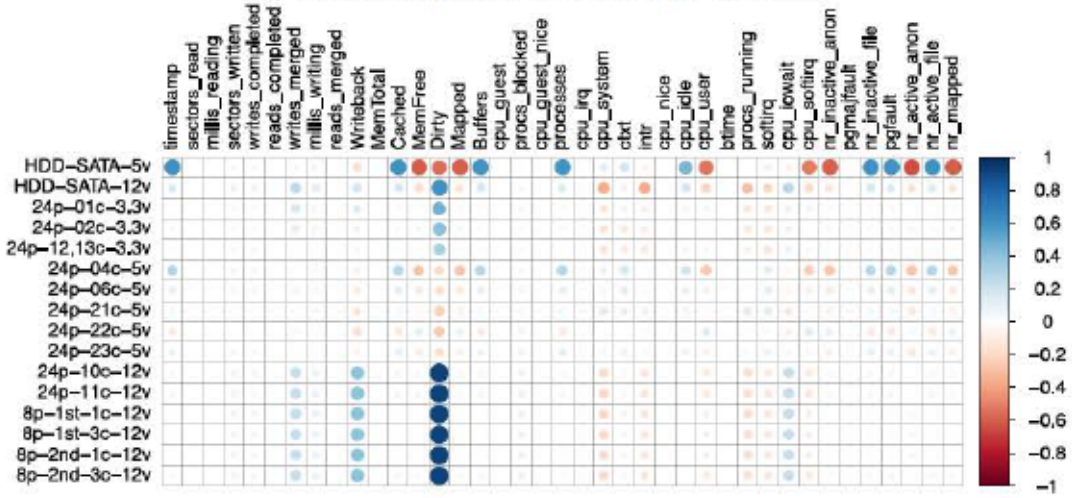
However, there is one aspect where the correlation plots with aggregated power usage (Figure 3.7) differ from the plots with per-line power consumption (Figure ??). The latter shows clearly two distinct groups of system metrics, where each group is correlated with different system components. This is due to the fact that some system metrics are only correlated with the power usage of the storage device (in our case, the hard drive) while the other group is correlated with the CPU and memory utilization. Therefore, the system metrics such as `sectors_written` and `writes_completed` are correlated with the disk's 5V power line (labeled HDD-SAT-5v), providing the dynamic power and dependent on the demand and utilization. On the other hand, `cpu_system` and `Dirty` show a high correlation with other system components, including CPU power usage during I/O activity.

Therefore, we can conclude that our analysis not only reveals which metrics are more correlated with specific system power lines during write I/O activity, but we can identify which parts of the system provide dynamic and static power for our use case. In our system, we were even able to run micro-benchmarks that stress specific system components to match each power line with a specific system component. However, note that mileage will vary depending on the electronic design of each system's motherboard. This knowledge can be useful to determine how much power

can be saved by making physical system components more power proportional[2] by reducing static power. Similarly, the dynamic power reveals the degree to which software can manage and reduce power consumption. In our systems, we observed the ratio between static and dynamic power for I/O to get as high as 50%.



(a) *Correlation plot for writes on a per-line basis.*



(b) *Correlation plot of derivative system metrics w.r.t. time, for writes on a per-line basis.*

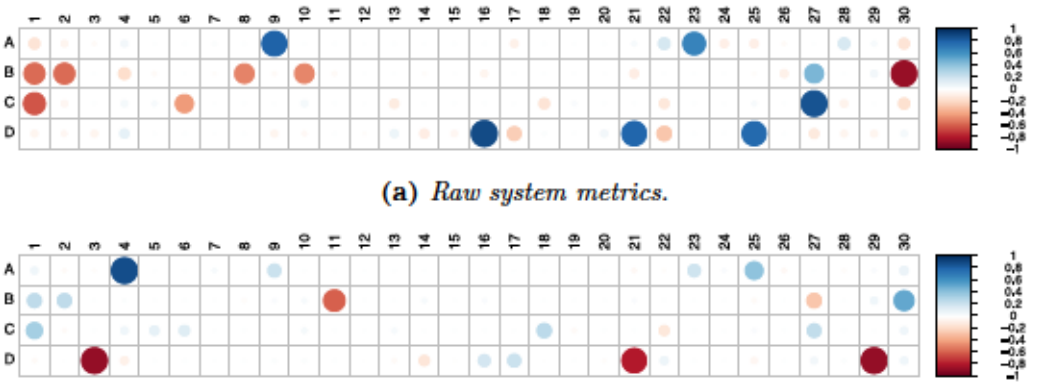
Figure 3.10: *Correlation plot between power usage and system metrics for a 4 GiB write.*

3.4.2 Analysis of read operations

In this section, we perform the analysis of read operations, following the methodology the same way we did for writes. Figure 3.11 shows correlations of the 120 system metrics that we used for write operations. The figure shows correlations with power

[2]Power proportionality is a metric that measures the degree to which the power used by a system is proportional to the required utilization [BH07].

usage of raw system metrics (top) and correlations for derivative system metrics w.r.t. time (bottom).



(a) *Raw system metrics.*

(b) *Derivative system metrics w.r.t. time.*

Figure 3.11: *Correlation plot between power usage and system metrics for a 4 GiB read.*

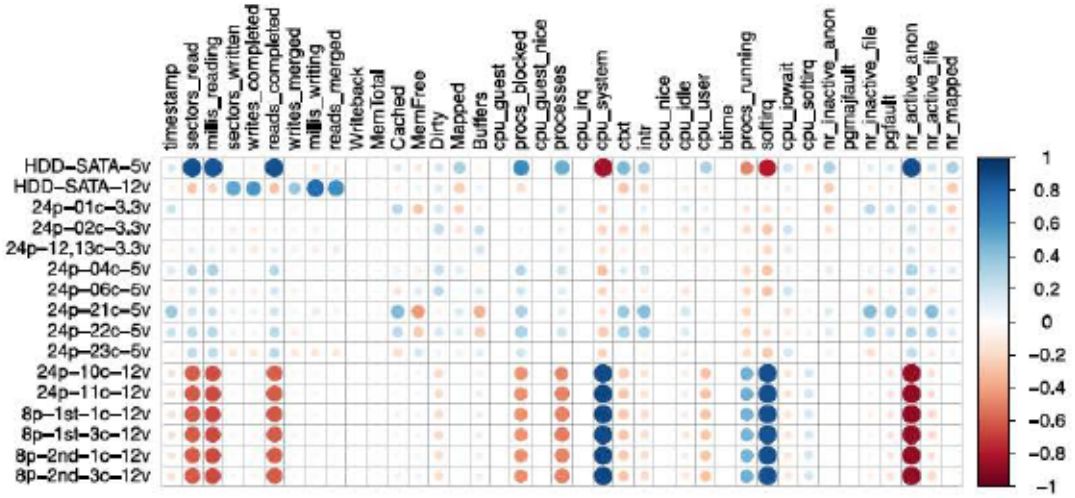
Table 3.2 lists the most significant system metrics by discarding the ones with a correlation below our empirically determined threshold of 0.75. This list reveals a few interesting metrics. As it was the case for writes, `cpu_system` shows a high correlation with CPU power. However, this metric does not allow us to decouple I/O power usage from power by activities not related to I/O. The system metric `cpu1_softirq` also turns out to be redundant since it reflects software interrupts on a particular CPU core. This correlation is circumstancial as subsequent experiments will probably generate interrupts on different CPU cores. On the other hand, the metric `softirq` shows a better correlation as it counts interrupts globally and thus solves the dependency on a particular subset of CPU cores. The system metrics `MemFree`, `nr_inactive_file` and `Cached` represent the free memory, inactive file mapped memory, and cached memory, respectively. Because their correlations are high when taking the derivative w.r.t. time, we consider these metrics good indicators of data movement. Note that `MemFree` and `Cached` may cease to be useful when the system is under memory pressure. As the system approaches the maximum memory and cache capacities, there would be no rate of change for either metric. However, this would not be the case for `nr_inactive_file` and `softirq`. We argue that a user trying to monitor the data movement activity should nevertheless look at all these metrics together.

Finally, we discuss the result of applying our methodology and analysis on a per-line basis for reads. The resulting plots are depicted in Figure 3.12. The top plot consists of correlations of raw system metrics while the bottom plot consists of correlations of the derived system metrics w.r.t. time. We arrive at a very similar conclusion to what we learned from write operations. The top plot reveals a very strong correlation between one group of system metrics with the hard drive's power lines, and a correlation between another group of system metrics with the rest of the system power lines. Therefore, the plot identifies three new relevant system metrics: `sectors_read`, `millis_reading`, and `reads_completed`. The reason why

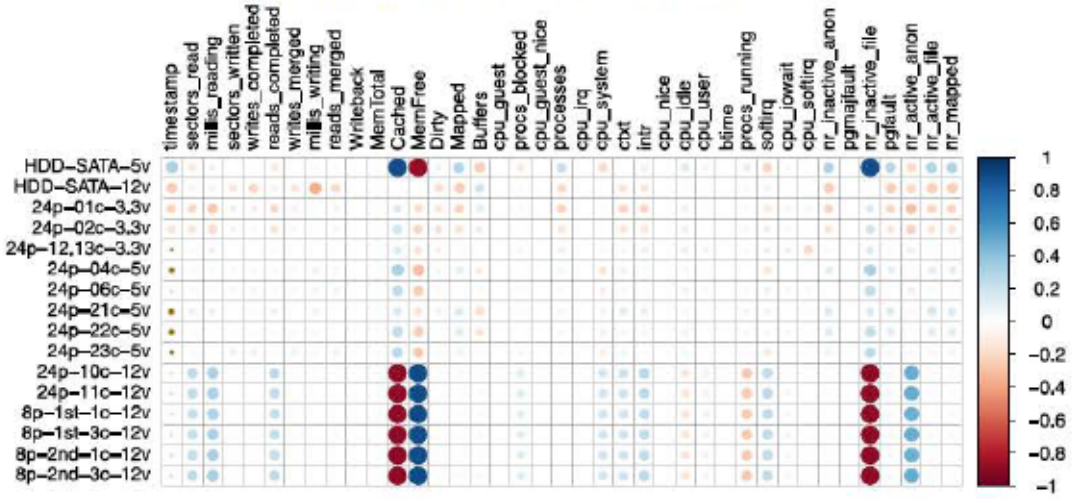
Table 3.2: *Correlation of system metrics to power for reads.*

Metric	Corr(data)	Corr(d/dt(data))	Corrplot Tile
`cpu_system`	**0.89**	0.18	D16
`MemFree`	0.05	**0.88**	A4
`nr_inactive_file`	-0.05	**-0.88**	D29
`Cached`	-0.05	**-0.87**	D3
`softirq`	**0.86**	0.24	C27
`nr_active_anon`	**-0.86**	0.5	B30
`cpu1_softirq`	0.79	-0.79	D21

these system metrics were not identified in our aggregated power plot is due to the fact that they are strongly connected to the HDD-SAT-5V power line, and their correlation with other power lines is much weaker. Similar to writes, we also demonstrate that a per-line power usage analysis can be useful to identify how certain system metrics are correlated with specific power lines. This knowledge can then be leveraged by I/O models and used on machines that are not physically instrumented with such powerful wattmeters.



(a) *Correlation plot for reads on a per-line basis.*



(b) *Correlation plot of derivative system metrics w.r.t. time, for reads on a per-line basis.*

Figure 3.12: *Correlation plot between power usage and system metrics for a 4 GiB read.*

3.5 I/O Power Modeling

We use the knowledge acquired during the analysis detailed in the previous sections to build a power consumption model. Based on the amount of data, I/O workload, and system performance, we present an energy consumption model that uses power regimes as building blocks. Note that, as was shown during the analysis, most I/O operations correspond to a specific power and performance regime. Hence, the power model described next has been thought out to be as simple as possible. However, we also demonstrated that writes are substantially more complex, and therefore we propose and compare two different models for writes in Section 3.5.1. We start by describing our general power model in a top-to-bottom fashion next.

The total energy consumption of the system can be expressed as the sum of energy consumed while performing purely computation and the energy consumed

doing I/O.

$$E_{Total} = E_{Compute} + E_{I/O} \tag{3.4}$$

This work concentrates mainly on studying the energy cost of performing I/O. For each component, there is a fixed amount of power that does not vary, regardless of the workload. Therefore, a distinction between *static* power and *dynamic* power is made. $E_{I/O}^{Static}$ represents the fraction of energy consumed by components and does not vary as a function of time or work performed. Dynamic power varies depending on resource utilization and workload patterns. Dynamic energy $E_{I/O}^{Dynamic}$ is defined as a function of I/O pattern, a weighted combination of $seqRead$, $randRead$, $seqWrite$, and $randWrite$, such that weights fulfill Equation 3.5.

$$w_{seqRead} + w_{seqWrite} + w_{randRead} + w_{randWrite} = 1 \tag{3.5}$$

$$E_{I/O} = E_{I/O}^{Static} + E_{I/O}^{Dynamic} \tag{3.6}$$

$$E_{I/O}^{Dynamic} = E_{Read} + E_{Write} \tag{3.7}$$

$$E_{Read} = w_{seqRead}\ \ E_{seqRead} + w_{randRead}\ \ E_{randRead} \tag{3.8}$$

$$E_{Write} = w_{seqWrite}\ \ E_{seqWrite} + w_{randWrite}\ \ E_{randWrite} \tag{3.9}$$

Each of $E_{seqRead}$, $E_{seqWrite}$, $E_{randRead}$ and $E_{randWrite}$ can be expressed as the energy E used during an experiment of duration T, where $P(t)$ is the instantaneous power for time instant t:

$$E = \int_0^T P(t)\,\mathrm{d}t \tag{3.10}$$

However, as we have observed in Section 3.3, I/O patterns can be discretized into distinct power regimes. Therefore, read and write power is expressed as a sum of discrete power levels, according to their respective power regimes. The energy for a sequential read I/O pattern is calculated as follows.

$$E_{seqRead}(Size, H) = E_{seqRead}^{Cache}(Size, H) + E_{seqRead}^{Disk}(Size, H) \tag{3.11}$$

$$E_{seqRead}^{Cache} = H\frac{Size}{Perf_{seqRead}^{Cache}}\ \times P_{seqRead}^{Cache} \tag{3.12}$$

$$E_{seqRead}^{Disk} = (1-H)\frac{Size}{Perf_{seqRead}^{Disk}}\ \times P_{seqRead}^{Disk} \tag{3.13}$$

Where $Size$ is the total amount of data that are being handled and H represents the expected hit ratio for modeled reads. $Perf$ is a matrix that contains performance values for either $Cache$ or $Disk$ accesses for each I/O pattern ($seqRead$, $randRead$, $seqWrite$, $randWrite$). P is a matrix that contains the average power values for disk and page cache accesses for each I/O pattern ($seqRead$, $randRead$, $seqWrite$, $randWrite$). P^{Cache} is used for modeling read hits and memory re-writes, while P^{Disk} is used for modeling disk access. These matrices represent the power regimes described in Subsection 3.3.1. The equations for random I/O are equivalent.

The values for these matrices are calculated from a set of micro-benchmarks. Each micro-benchmark measures all four I/O patterns under different conditions, such as the file being fully cached in the page cache for populating $Perf^{Cache}$ and P^{Cache}. And the opposite, all dirty buffers flushed and cleared page cache for obtaining $Perf^{Disk}$ and P^{Disk}. All possible combinations for populating the power matrix P are shown in Table 3.3. The micro benchmarks were run using the benchmark tool fio [Axb]

Table 3.3: *Possible combinations for power matrix P. Left column represents power values for disk access, right column for memory access. Rows are different I/O access patterns.*

	Disk	Cache
Sequential Write	$P^{Disk}_{seqWrite}$	$P^{Cache}_{seqWrite}$
Random Write	$P^{Disk}_{randWrite}$	$P^{Cache}_{randWrite}$
Sequential Read	$P^{Disk}_{seqRead}$	$P^{Cache}_{seqRead}$
Random Read	$P^{Disk}_{randRead}$	$P^{Cache}_{randRead}$

In our case, the aggregated power usage of every component C is modeled as

$$P^{Disk}_{seqRead} = \sum_{C}^{Components} P^{Disk}_{seqRead}(c) \tag{3.14}$$

This is done because we had a powerful instrument which is capable of obtaining per-component power readings. However, we do not expect any user of the model to perform this kind of instrumentation in order to obtain a valid model for their hardware. For this reason, our model is designed in a way that one can measure the aggregate power consumption for each regime (cache access, disk access, different I/O patterns) in order to build the aforementioned power matrix P.

The energy for write I/O is computed in a different manner. While the model for reads is built using performance regimes based on different I/O access patterns such as reading from memory or reading from disk, these regimes occur one at a time, and are modeled accordingly. That is, either the system is in a state where data is being read from the page cache, or the system is in a state where data is being loaded from disk. However, during writes, the system can be performing a background write, while a user space process can be adding dirty data to the page cache at different rates, resulting in different regimes for the same write access pattern. Therefore, we distinguish between the energy required to flush dirty data from memory to disk, E^{Disk}_{Write}, and the energy required for the user space process to write this data to the page cache, E^{Cache}_{Write}.

$$E_{Write}(Size, H) = E^{Disk}_{Write} + E^{Cache}_{Write} \tag{3.15}$$

Note that E^{Disk}_{Write} is not only the energy consumed by the storage device, but includes the energy consumed by main memory, CPU, and other components in order to move the data to storage. P^{Disk}_{Write} is the average power consumption for background writes when no other activity is being carried out (the blue regime in Figure 3.4), while P^{Cache}_{Write} is the power consumption for writing data to the page cache.

$$E^{Disk}_{Write}(Size, H) = \frac{Size}{Perf^{Disk}_{Write}} \times P^{Disk}_{Write} \tag{3.16}$$

$$E^{Cache}_{Write}(Size, H) = \frac{Size}{Perf^{Cache}_{Write}(H)} \times P^{Cache}_{Write} \tag{3.17}$$

There are two alternatives for modeling P^{Cache}_{Write}. For large writes, an average value can be used and is sufficient, resulting in a simpler formula. However, write power will vary over time depending on the state of global dirty memory. In addition, we found that different storage devices will have a different impact on CPU and memory resource utilization. A more accurate way of modeling P^{Cache}_{Write} is detailed in Section 3.5.1. For smaller writes or when the modeled power over time is required, the detailed model is preferred. We evaluate and compare the effectiveness of both models in a detailed manner in Section 3.6.2.

Furthermore, we distinguish between two cases: the page cache is being rewritten and new data is being written to the page cache, which requires allocation. While this may seem a far fetched detail, we observed between a 50% and 60% difference in performance between both cases. Thus, in the case of writes, H represents hits in the page cache, meaning no new page cache entries need to be allocated for the data.

$$Perf^{Cache}_{Write}(H) = H \times Perf_{allocated} + (1 - H) \times Perf_{unallocated} \tag{3.18}$$

Where $Perf_{allocated}$ and $Perf_{unallocated}$ represent the performance for rewriting data in the page cache, and writing new data to the page cache, respectively.

3.5.1 Modeling writes for a storage device

While the throttling mechanisms detailed in Section 3.3.2 are directly related to power and performance regimes, we found that the file system and storage device internals will also influence transition and duration of regimes. The analysis of write operations has revealed distinct power regimes and their relation to system performance counters, but we found no practical way of predicting when transitions between regimes are going to happen. Similarly, we found no way of predicting the duration of each regime. When testing with different storage devices and file systems, we found these variables to vary greatly and unpredictably. Each storage device will

produce very different data movement patterns not only due to differences in latencies and throughput, but also due to the way different devices process block requests internally. For these reasons, we decided to create a model for writes that abstracts a particular file system and storage device.

While an average power value can be used for modeling writes, just as described for reads, this only works well for large file writes. For smaller file writes, errors can be too large to be ignored. This is because, as depicted in Figure 3.4, the state of dirty buffers will have a great impact on whether write operations block or not, and for how long. This is strictly related to CPU consumption. Therefore, a file write that starts when dirty buffers are empty will exhibit a very different behavior and power consumption when compared to a file write that starts when dirty buffers are almost full.

These data movement patterns, observed via the aggregated dirty buffer size, represent how user space applications performing I/O are using components other than the storage device itself, and have a direct impact on the CPU and memory power consumption. Thus, for modeling writes we used an approach which takes into account the state of dirty buffers. A dirty memory time series, data that can be obtained from */proc/meminfo* in Linux with negligible overhead, holds the information for the different regimes. Write power consumption is strictly related to the rate at which memory is being added to the page cache and moved to disk. We combine the dirty time series γ with the power usage time series π to create a power consumption model M based on a storage device. Using this information, no discretization into power regimes is required. While the aforementioned regimes are still valid, we can use data movement information provided by γ to predict write power consumption, thus providing a smooth and more accurate transition between regimes.

Using the dirty time series γ and the power usage time series π, a power consumption model for a storage device is created as follows.

First, the rate at which data are being added to the page cache ρ is calculated from γ. The rate of change of dirty data is $\frac{d\gamma}{dt}$, where dt is the dirty memory sampling time. When $\frac{d\gamma}{dt}$ is zero, the operating system is throttling write operations to match the disk write-out bandwidth. Negative values often indicate that the process is blocked, and the background disk flusher threads doing write-back are decreasing the amount of dirty memory.

In order to accurately represent the rate ρ of data movement to the page cache, we consider the minimum values of the derivative to be the point at which the dirty memory is being flushed to disk and no new memory is being dirtied. Therefore, we offset all values of $\frac{d\gamma}{dt}$ accordingly so that $\rho = 0$ when no data is being added to the page cache. Thus, $\rho = \frac{d\gamma}{dt} - min(negative\_values)$. We define $negative\_values$ as the negative values of the derivative: $negative\_values = \frac{d\gamma}{dt} < 0$.

During this process, we infrequently encountered measurement errors in some experiments that caused $\frac{d\gamma}{dt}$ to contain one negative value that is too large. If only one value of the derivative is such an outlier, this gives the false impression that the disk can write dirty data to disk faster than it physically can. This causes the offset mentioned previously to contain a large error. To remove these outliers, we exclude

negative data points of $\frac{d\gamma}{dt}$, which are below the threshold ν, defined as follows.

$$\nu = mean(negative\_values) - stdev(negative\_values) \tag{3.19}$$

Having obtained the time series ρ of the rate at which data is being added to the page cache, we use the corresponding power time series π to create a model that outputs power usage when provided with a dirty memory time series. For that purpose, we perform a linear least squares regression to model $power \sim throughput$. However, ρ is a time series derived from the data of a file write, where write throughput to the page cache does not increase linearly over time, but oscillates. Our goal is to build the regression using power data as a function of a linearly increasing throughput. This information is extracted from ρ and π. We define $l(t)$ as the function that, given the time series t, returns the indices for linearly increasing values of t. Therefore, it is possible to select the corresponding power values from π for linearly increasing values of write throughput ρ. The linearly increasing values of write throughput, $\rho_{l(\rho)}$, are obtained by selecting $l(\rho)$ from ρ. The corresponding power values, $\pi_{l(\rho)}$, are obtained by selecting $l(\rho)$ from π. Finally, the model for $power \sim throughput$ is built by doing a linear regression on $\pi_{l(\rho)} \sim \rho_{l(\rho)}$. This procedure is described in Algorithms 1 and 2. The corresponding regression is depicted in Figure 3.13.

Algorithm 1 Write power model

```
1: procedure ModelPowerDirty(π, γ)   ▷ Model power as a function of dirty buffer size.
2:     ρ ← DirtyToThroughput(γ)                                   ▷ See Algorithm 2
3:     throughput ← list()
4:     for i in l(ρ) do
5:         throughput.add(ρ_i)
6:     end for
7:     power ← list()
8:     for i in l(ρ) do
9:         throughput.add(π_i)
10:    end for
11:    P_Write^Cache ← LinearRegression(power ∼ throughput) return P_Write^Cache
12: end procedure
```

Our model has been tested to predict power consumption with an accuracy of under 8%, as demonstrated in Section 3.6. While our model for writes has been derived from the data acquired when writing a 20GB file, we tested how the accuracy varies when increasing experiment duration. Varying the write experiment length from one second to 150 seconds, in steps of 0.1 seconds, we built around 1500 instances of the model, and evaluated the resulting error. Figure 3.14 depicts how the mean absolute percentage error changes as the model is built based on experiments of increasing duration. It is worth noting that for models derived from very short experiments there were error data points well above 100%, but since this was expected, we omitted them for the sake of clarity so we could preserve the scale and discern-ability of these graphs.

Algorithm 2 DirtyToThroughput

procedure DIRTYTOTHROUGHPUT(γ) ▷ Takes dirty buffer size time series as input, ouputs memory write throughput.
  $neg\_vals \leftarrow list()$
  for all $i \subset \frac{d\gamma}{dt}$ such that $i < 0$ **do**
    $neg\_vals.add(i)$
  end for
  $\nu \leftarrow mean(neg\_vals) - stdev(neg\_vals)$
  $\rho \leftarrow list()$
  for all $i \subset neg\_vals$ such that $i \geq \nu$ **do**
    $\rho.add(i)$
  end for
  $\rho \leftarrow \rho - min(\rho)$
return ρ
end procedure

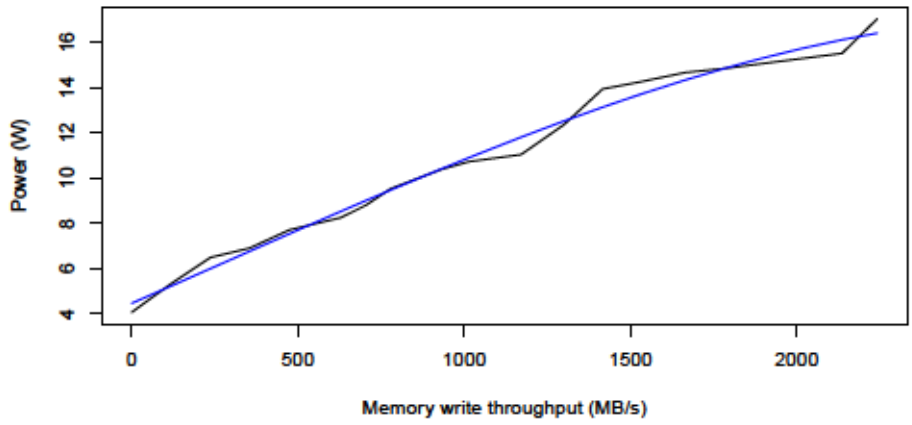


Figure 3.13: *Resulting write power model. Power consumption as a function of write throughput to memory.*

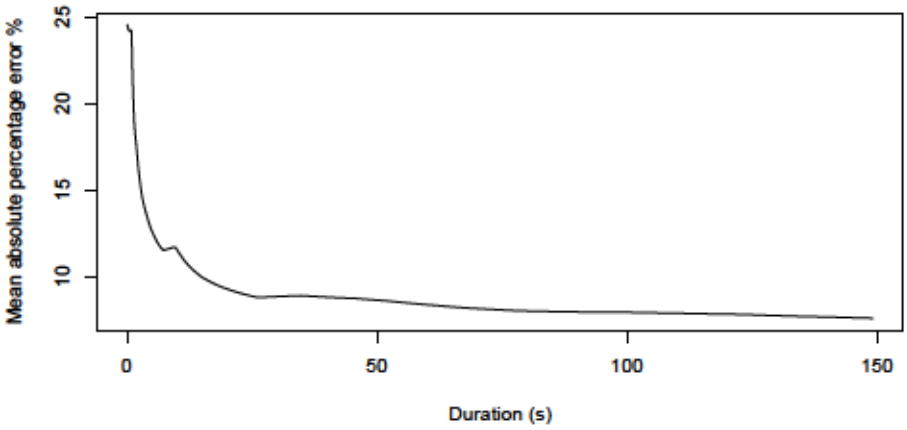


Figure 3.14: *Mean absolute percentage error shows decreasing errors.*

3.6 Evaluation

The setup used for our evaluation has been detailed in Section 3.2.2, and uses a combination of several storage devices. As previously mentioned, we found the differences in these devices to have a huge impact on power and CPU usage. Even between both hard disk drives, where we repeatedly measured a peak performance difference no greater than 12%, data movement patterns varied greatly, especially during writes, as detailed in Section 3.5.1.

For evaluating the accuracy of the models described in Section 3.5, we executed a series of micro benchmarks that stress the system's storage I/O layer, using different I/O access patterns. Every experiment was run at least five times. We took the average of every measured power and performance value and computed the standard deviation. Also, note that these evaluations represent the *dynamic* energy $E_{dynamic}$. This is due to the fact that adding the constant value of E_{static} to the results would mistakenly seem to reduce the modeled error and make the differences shown in the graphs more difficult to read. We observed E_{static} to be around 50% of $E_{dynamic}$ for typical I/O workloads.

3.6.1 Read workload evaluation

Figure 3.15 (left hand side) depicts the measured and predicted values for sequentially reading a 20GB file that was not cached in memory. We modeled the sequential read access pattern using Equations 3.11 and 3.13, and obtained an error of 6.9% for HDD1, 3.4% for HDD2, and 5.5% for SSD. The relative standard deviation for these power measurements was around 1%.

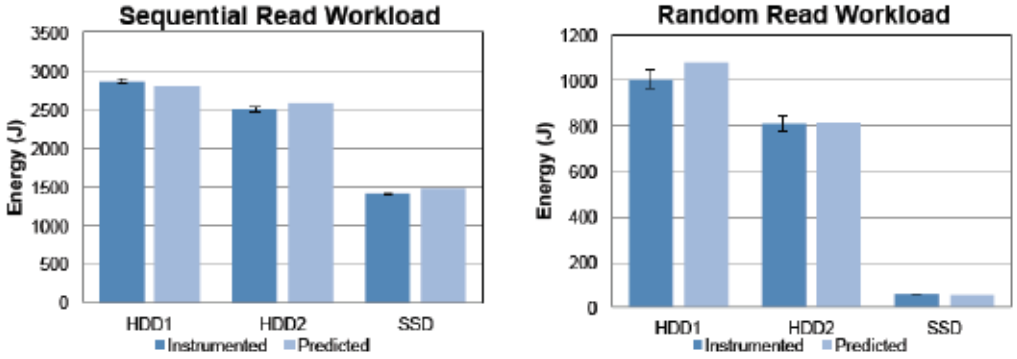


Figure 3.15: *The figure plots a 20GB sequential read (left) and a 100MB random read (right).*

In Equation 3.14 we assume, based on our observations made during the analysis of read operations, that a change in the amount of data present in the page cache, results in proportional change of power used from each power regime. We test this hypothesis and the validity of our model with the following experiment. Figure 3.16 depicts sequential 4GB file reads for varying amounts of the file in memory. The red squares represent 6 different experiments, where a hit ratio of 0% means all of the

file data had to be moved from disk to memory, and a hit ratio of 100% means that all of the file data was present in the page cache. We modeled these access patterns using Equations 3.11, 3.12, and 3.13, varying parameter H from 0 to 1 in 0.2 step intervals in order to model page cache hit ratio. For the predicted values from 0% to 100%, the normalized root mean square error is 1.3%.

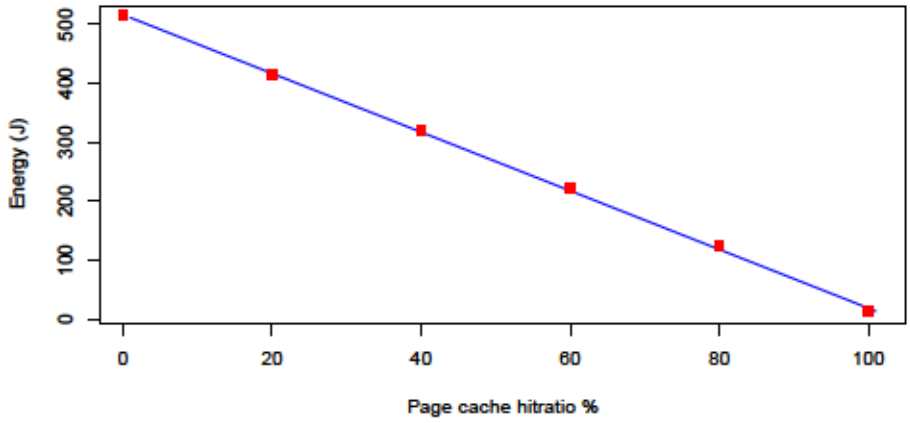


Figure 3.16: *4GB Sequential read varying the page cache hit ratio from 0% to 100%. Red squares represent measured amount of energy. The blue line represents the energy consumption as predicted by the model.*

Figure 3.15 (right hand side) depicts the measured and calculated values for a random read of a 100MB file that was not cached in memory. This is modeled using Equations 3.11 and 3.13, but using a random access pattern instead of sequential access. We obtained an error of 7.1% for HDD1, 0.6% for HDD2, and 8% for SSD, while the relative standard deviation for HDD1, HDD2, and SSD was of 1.2%, 2.5%, and 5.6% respectively.

3.6.2 Evaluation and comparison of write models

In Section 3.5 we presented to ways to model writes. One that is based on the average write power, and a more sophisticated model which uses the state of dirty buffers to more accurately model the data movement within the I/O and memory hierarchy. Therefore, we evaluate and compare the effectiveness of both models.

As is expected, for writes of a sufficiently large duration, the simple approach of using an average write power works well. On the other hand, writes of a smaller duration will cause write power to fluctuate significantly with each regime transition, depending on the amount of dirty memory (as was detailed in Section 3.3.2). Therefore, we compute the mean absolute percentage error for writes of different duration for both models.

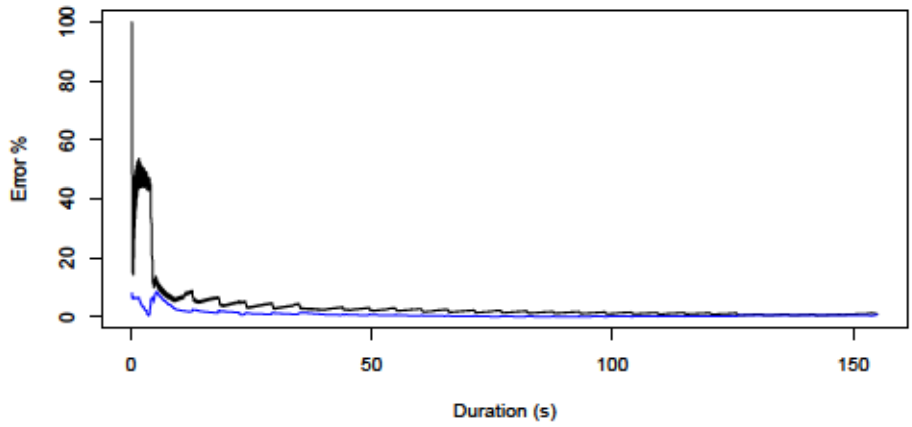


Figure 3.17: *Errors for short vs large file writes are compared. The model where P_{Write}^{Cache} is an average power value is the upper line, shown in black. The write model described in Section 3.5.1 is the bottom line, shown in blue.*

Figure 3.17 shows the error for both models for writes up to 150 seconds in length. As it can be clearly seen, small writes suffer from a large error when an average power value for P_{Write}^{Cache} is used. When the amount of data written to the page cache approximately reaches the dirty memory limit, both models exhibit a very similar error. This is due to the fact that the operating system's dirty memory throttling mechanisms will cause write throughput to vary substantially over time, and an average value suffers from being unable to adapt to the current state of dirty buffer sizes. However, as more data is written over time, an average value works well, as was expected. Note that Figure 3.17 compares both P_{Write}^{Cache} models by calculating and comparing only E_{Write}^{Cache} (Equation 3.17), not E_{Write}, where the error would be smaller.

3.6.3 Write workload evaluation

We use the more sophisticated write model, detailed in Section 3.5.1, to evaluate the prediction of power and energy consumption for several write micro benchmarks next. Figure 3.19 (right hand side) depicts the measured and calculated values for sequentially writing a 20GB file that was not cached in memory. We modeled this access pattern using Equations 3.15, 3.16 and 3.17 using $H = 0$, and the model described in Section 3.5.1. We obtained normalized standard errors of 6.9% for HDD1, 5.6% for HDD2, and 3% for SSD, while the relative standard deviations for power measurements were below 1% for all three devices. Using this write model, in addition to the total amount of consumed energy, the power over time is also obtained, which might be of interest for the user. Figure 3.18 depicts modeled power measurement over time for writing 20GB of data to the page cache. The normalized root

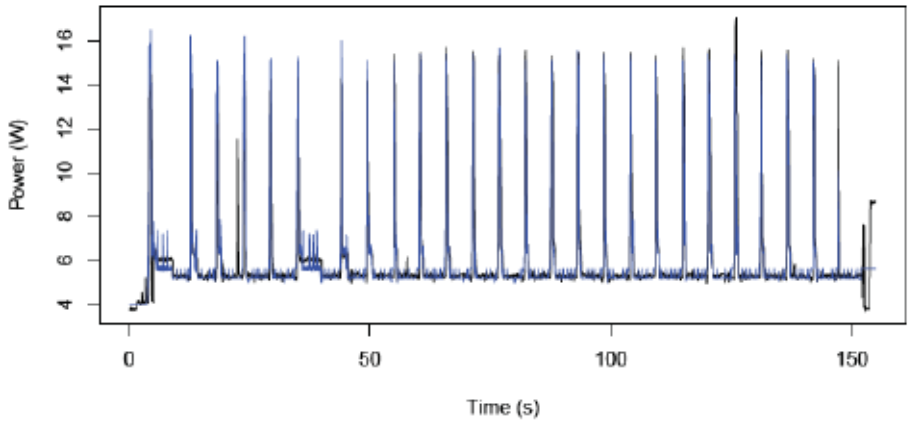


Figure 3.18: *20GB Sequential write. Measured power consumption (black) and predicted power consumption (blue) based on dirty memory information.*

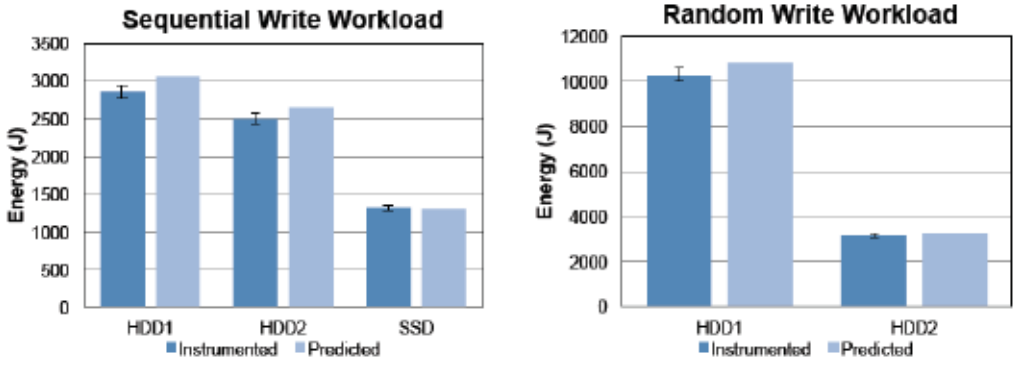


Figure 3.19: *The figure plots a 20GB sequential write (top) and a 4GB random write (right).*

mean square error for these time series was calculated to be 8%.

Figure 3.19 (right hand side) depicts the measured and calculated values for a random write of a 4GB file that was not cached in memory. The SSD was not included in this figure because the performance for random writes was much higher compared to that of the hard disk drives, resulting in very low energy consumption in relation to the one caused by hard disks, rendering its columns unreadable (values were 23.799J measured, 24.6J predicted). Equations 3.15 and 3.16 were used with a random access pattern, yielding normalized standard errors of 5% for HDD1, 4% for HDD2, and 3.3% for SSD. Relative standard deviations for these experiments were 2.6%, 2.5%, and 0.2%, respectively.

3.6.4 Mixed workload evaluation

Finally, we run mixed read and write workloads using different access patterns. We performed these mixed workloads using Jens Axboe's Flexible I/O tester [Axb] on HDD2. Figure 3.20 compares measured and predicted energy for three mixed workloads.

Workload W1 consists of a strided read of 128MB blocks and a 256MB stride, reading 2GB of data in total. Then, the file is re-read sequentially (4GB), followed by a sequential file re-write, followed by appending 4GB of data, followed by two threads randomly mixing reads and writes (50%) with an I/O size of 4k and an aggregated data transfer of 2GB. This was modeled using a 2GB sequential read (Equations 3.11 and 3.13) for the strided read, another sequential read using $H = 0.5$ for the re-read. The re-write and append is modeled by employing Equations 3.15, 3.16 and 3.17 and using $H = 1$. The threads doing random reads and writes are modeled using equations for random read and writes, with 1024GB sizes each. Compared to measured energetic consumption, the sum of these predicted values had a normalized standard error of 6.8%.

Workload W2 consists of a 4GB random write, followed by a strided read and a file re-read. Then the whole file is re-written, and another 4GB of data are appended to the file. This is modeled using Equations 3.15 and 3.16 for a random access pattern. The strided read is modeled using Equations 3.11 and 3.13, and for sequential read the hit ratio parameter H is set to $H = 0.5$ for the re-read. The re-write and file append can be modeled as a 8GB sequential write (Equations 3.15, 3.16 and 3.17 using $H = 1$). The prediction had a normalized standard error of 1%.

Workload W3 consists of a 4GB random write, followed by appending another 4GB of data, followed by two threads randomly mixing reads and writes (50%) with an I/O size of 4k and an aggregated data transfer of 2GB. This workload was modeled using Equations 3.15, 3.16 and 3.17 using random and sequential access patterns for the random write and append, respectively. The two threads were modeled as in Workload W1. Compared to our measured energy consumption, our prediction had a normalized standard error of 2.8%.

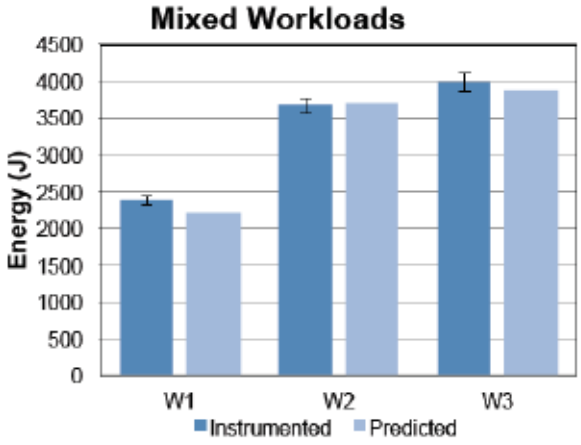


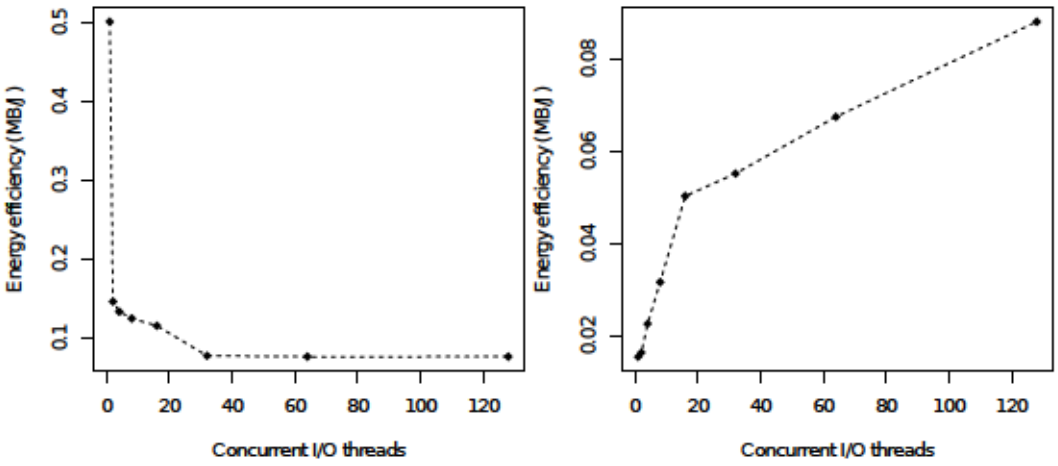
Figure 3.20: *Comparison of measured energy with model predicted values for three workloads that mix reads and writes using different I/O patterns.*

3.7 Modeling multi-threaded I/O

Up to this point, only single-threaded file access has been considered. This is sufficient for mico-benchmarks, but more realistic macro-benchmarks perform multi-threaded I/O on a collection of files.

There are several important challenges when modeling multi-threaded I/O. First, the access pattern may evolve over time based on factors such as the dataset being cached in memory. Second, and related to the previous point, it is now possible that one thread reads data that another thread just read or wrote, which will result in read hits even though this thread never read the data. Third, increasing the number of concurrent in-flight I/O operations by spawning multiple threads or processes will have an impact on power consumption due to the fact that more of the machine's resources are being utilized.

The first and second points can be resolved by dividing the application into distinct regimes based on their access pattern, and modeling the application's hit ratio H for each regime. However, the issue of higher resource utilization when varying the number of threads is more complex due to the sheer amount of variables involved: type of access pattern for each thread, I/O size, number of concurrent threads, type of storage device, etc. To model multi-threaded I/O, we consider the following factors: the *scalability* of the *energy efficiency* when increasing the number of concurrent I/O threads for a specific access pattern and the storage device. These scalability curves can be calculated using the expected storage device's IOPS, total data set requirements, and power usage for each access pattern. For instance, a workload dominated by reads will show poor energy efficiency scalability on a conventional hard disk, as depicted in Figure 3.21a, while a workload dominated by random writes will offer better scalability, as depicted in Figure 3.21b. This is due to the fact that the operating system will perform *backmerge* operations on the block layer that will improve performance and energy efficiency as the number of



(a) *Random read scalability.* **(b)** *Random write scalability*

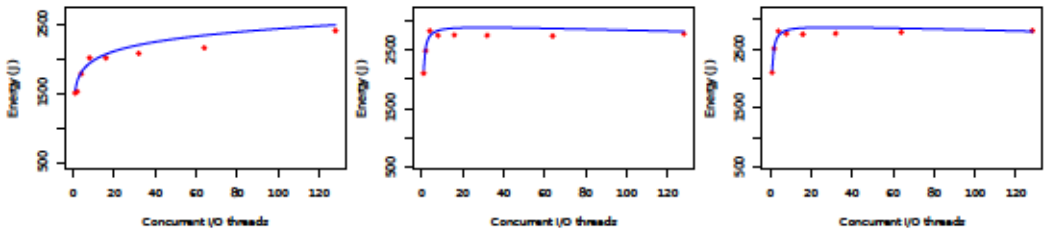
Figure 3.21: *Scalability of energy efficiency for read and write workloads.*

concurrent I/O threads doing random writes increases. Each curve is modeled using simple linear regression.

3.7.1 Macro benchmark evaluation

We build upon our micro benchmarks to evaluate different I/O intensive scenarios using Filebench [Wil08]. Filebench includes a series of macro benchmark workloads that model a mail server (*varmail*), a web server (*webserver*), and a proxy server (*webproxy*). We scale each workload to up to 128 I/O threads.

varmail. The mail server workload consists of a number of threads concurrently operating on a file set of 1000 files. Each thread iterates over the following three phases: First, one file is picked out of the dataset and truncated to zero, then data is written. Second, another file file is picked out of the dataset and is read sequentially. Then, data is appended to the file. Finally, a third file is read sequentially. All write operations are 16KB in size, and are followed by a $f$sync() call. The average file size is 16KB, but due to the appends and re-writes, will converge to an average file size of 32KB. In this scenario, dirty data are kept to a minimum, as all threads will quickly iterate over $f$sync() calls after every 16KB write. Moreover, data will be slowly read into the page cache until most or all data are cached. Note that while files are being re-written continuously, the new data will be cached in memory. Therefore, this workload will first combine random writes with small sequential reads, and at a certain point it will be mostly read cache hits, with small random writes running in the background. The read to write ratio is 1:1. We model this workload as the sum of two different regimes, A and B. Regime A is modeled as a combination of small sequential reads from disk and small random writes to disk, while regime B is modeled as read page cache hits, and small random writes to disk. Regime A will be $\frac{nfiles \times 32KB}{Perf^{Disk}_{Read}}$ seconds long, and the rest of the execution is spent in Regime B. Our model predicts that the energy efficiency (MB/J) increases as the number of



(a) *varmail workload.* **(b)** *webserver workload.* **(c)** *webproxy workload.*

Figure 3.22: *Macro-benchmark model evaluation for Filebench's varmail, webserver, and webproxy workloads. Red is measured energy consumption for a 1 minute Filebench run, blue is predicted energy consumption.*

concurrent I/O threads increase.

webserver. The web server workload performs in each thread many sequential reads, followed by a small (16KB) write append. The write accesses always the same file, simulating a log-based file write. Files belonging to the read set have a mean file size of 16KB. We model this workload as two different regimes, *A* and *B*. During regime A, files are being read sequentially from disk, until most of the data set is cached. From that point on, regime B reads hit the page cache and writes are periodically synched to disk.

webproxy. The web proxy server is very similar to the web server, where each thread performs many small sequential reads. The main difference is that there is no log file, and every 5 file reads, one of the files is re-written. The file set consists of 1000 files with a mean size of 16KB.

Each workload is run with 1, 4, 8, 16, 32, 64, and 128 threads. We model each workload as the sum of their respective regimes. For each regime, we use the multi-threaded I/O models described earlier that, based on the concept of *scalability of energy efficiency*, are able to predict energy consumption for a given access pattern and number of threads. As demonstrated in the results in Figures 3.22a to 3.22c, we find that our model not only predicts which applications benefit from increased multi-threading, and which do not, but is also capable of accurately predicting total energy consumption. Both webserver and webproxy workloads show that as the number of concurrent I/O threads increase, the energy efficiency of the workload decreases, while the varmail workload offers better scalability.

3.8 Summary

In this chapter, we demonstrate that through the collection of system metrics, data exploration and data analysis, it is possible to extract useful information from workloads, and reason about this information to understand in detail how workloads make use of specific system resources. This data-driven process, in turn, provides the necessary knowledge to build models that perform accurate predictions. Our case focuses on studying the power usage and energy consumption of data movement caused by I/O intensive workloads. We analyze system metrics data to discover

and learn about system transitions between different power and performance regimes under various access patterns, and leverage this knowledge to build power usage prediction models. More precisely, we demonstrate the validity of our approach with the following contributions:

First, a methodology for collecting and exploring data, which we use to discover power and performance regimes in I/O operations.

Second, we have detailed our combined hardware and software instrumentation for collecting the data that is later analyzed as power usage and system metrics time series.

Third, a methodology for extracting useful information from the power usage and system metrics time series. We demonstrate how this methodology is capable of automatically identifying system metrics which correlate with different power and performance regimes.

Fourth, we provide power models that are capable of predicting energy consumption for I/O workloads that combine sequential and random reads, writes, and multithreaded I/O. These models are evaluated using *f*io micro benchmarks and *F*ilebench macro benchmarks.

Fifth, We provide insights into how the data movement inside a single machine has big implications on how energy is used, even for simple operations such as sequential POSIX write. The write model uses information provided by the operating system to derive data movement patterns across the memory hierarchy and predict power usage over time. We compare the detailed write model with a simple straw-man approach to demonstrate the impact of data movement across the memory hierarchy and the effectiveness of our model.

Chapter 4

CPU-level data I/O energy efficiency and optimizations

When contemplating the power usage within a node, the CPU has always been the component that ranks among the biggest sources of power consumption. As such, a lot of effort has been put by hardware vendors into making it as power proportional as possible. Therefore, CPUs have long employed techniques which provide dynamic power and performance. Modern CPUs even feature a maximum performance that is dynamic. Processors employ techniques that take advantage of the thermal headroom to improve performance by opportunistically adjusting its voltage and frequency. However, the limit to how high and for how long the performance can be sustained at the highest levels is based on thermal and energy constraints. In this context, optimizing not just for lower power consumption, but also for lower CPU temperatures becomes a desirable goal.

In light of the increasingly dynamic nature of modern processors, vendors enable mechanisms for the operating system to manage power and performance. These dynamic features result in systems software such as kernel and runtimes having to manage the performance and power trade-offs of the CPU. While operating systems are designed with power-awareness in mind, each component is optimized for the raw performance within its own domain. For instance, the CPU management module is usually unaware of the status of other parts of the system. There is little cooperation between software layers to optimize power usage and energy efficiency, especially where it concerns computation and storage I/O.

In this chapter, we explore different strategies that aim to increase coordination between operating system modules that target computation and I/O. Our main goal is to improve the CPU energy efficiency, to lower the average CPU temperature, and do so without sacrificing performance. We propose various micro-processor level I/O efficiency improvements, and detail several strategies that target energy efficiency

improvements and thermal imbalance reduction. Our contributions can be summarized as strategies that are effective on three different levels. First, thermal imbalance reduction. Second, I/O aware CPU performance management. And finally, I/O- and thermal-aware task placement.

This chapter is structured as follows. Section 4.1 shows motivating experiments for addressing the two main issues addressed in this chapter: thermal imbalance and lack of CPU I/O awareness. Section 4.2 describes the design, implementation and evaluation of a novel strategy for reducing CPU cores temperature variation. While we successfully reduce temperature variation, we note that this strategy has a negative impact on performance and increases performance variation.Subsequently, we propose two novel techniques that address the shortcoming of this naive approach. Section 4.3 describes a strategy for exploiting CPU-level heterogeneity for improving the energy efficiency of workloads that combine I/O and computation. Section 4.4 presents a novel strategy for increasing coordination between the software layers that manage storage I/O and CPU performance states. We discuss the design of a runtime and evaluate our solution.

4.1 Motivation for addressing CPU-level inefficiencies

The purpose of this section is to introduce two motivating experiments that address two issues present in modern processors. The first issue is related to the growing amount of variability in modern processors. Modern systems feature more than one CPU package and an increasing number of cores per package. Uneven heat distribution within a single node causes a thermal imbalance among cores and packages, that in turn results in heterogeneous and unpredictable performance. We perform a motivating experiment which demonstrates this issue.

The second issue addresses the lack of coordination and cooperation between two layers of the operating system: CPU management and the storage I/O stack. The operating system layer responsible for managing the CPU boosts performance when required by running tasks at the expense of increased power consumption. We perform an experiment that consists in running an I/O intensive workload, and study how the CPU manages power and performance.

4.1.1 Motivation for addressing CPU-level thermal imbalance and heterogeneity

In this section we present an experiment that demonstrates CPU-level thermal imbalance and heterogeneity when the CPU is subject to high utilization for a prolonged period of time. We demonstrate this effect by running a CPU-intensive workload. For stressing the CPU we run DGEMM, a double precision matrix multiplication algorithm leveraging Intel's Math Kernel Library, on each core. Figure 4.1 shows how the temperature varies over time. As expected, the CPU core temperatures rise significantly within seconds. More interestingly, the temperature difference between cores residing on each package becomes greater over time, even though both CPUs

are identical and feature the same Stock Keeping Unit (SKU). This simple experiment demonstrates that the thermal imbalance is both cores and package wide. Blue and red colors are used to distinguish the core temperature from different packages. This variation may be caused my multiple factors, ranging from variations in the manufacturing process [KKK+08], to the flow of heat within the system enclosure [ZOMM+15].

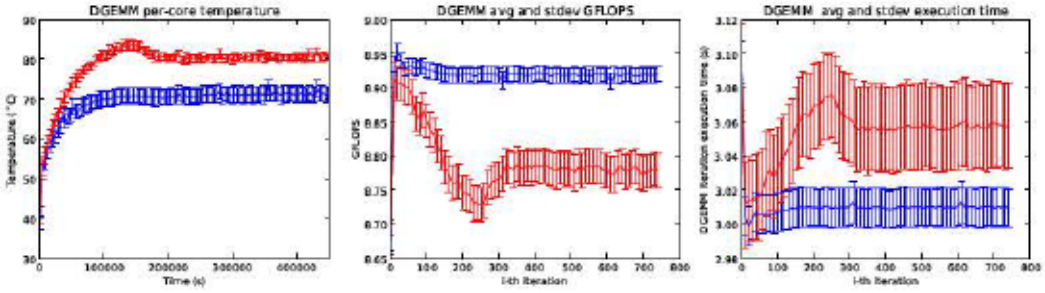


Figure 4.1: *Variation of CPU cores temperature (left) causes performance variability and degradation visible as measured GFLOPS (middle), and runtime (right) running a CPU-intensive workload. Cores belonging to the same package have been colored the same.*

The conclusion that can be drawn from this observation is that modern processors can no longer be regarded as homogeneous, as heterogeneity is manifested even within a node consisting of identical CPUs. Consequently, the next sections will detail novel techniques that reduce thermal variation and exploit the heterogeneity to improve the energy efficiency in Sections 4.2 and 4.3.

4.1.2 Motivation for addressing CPU energy efficiency during I/O workloads.

In this section we discuss an experiment that shows that the CPU management algorithms are inefficient during I/O workloads. We perform a sequential write of a 4GB file on an ext4 file system and observe power and performance usage of the CPU. Figure 4.2 depicts power and performance metrics during the execution of the I/O task. We observe that, despite the fact that the execution of write operations consist of a series of different power and performance regimes, the CPU runs most of the time at the highest performance and power-demanding states, similar to how the CPU would behave when running a computationally-demanding workload. We hypothesize that a lower frequency will lower power usage without impacting I/O performance, and that the current CPU performance management algorithm can be made more efficient by providing I/O-awareness. This motivates us to explore mechanisms for adaptively changing CPU performance states in coordination with storage I/O phases in order to improve the CPU energy efficiency during I/O in Section 4.4.

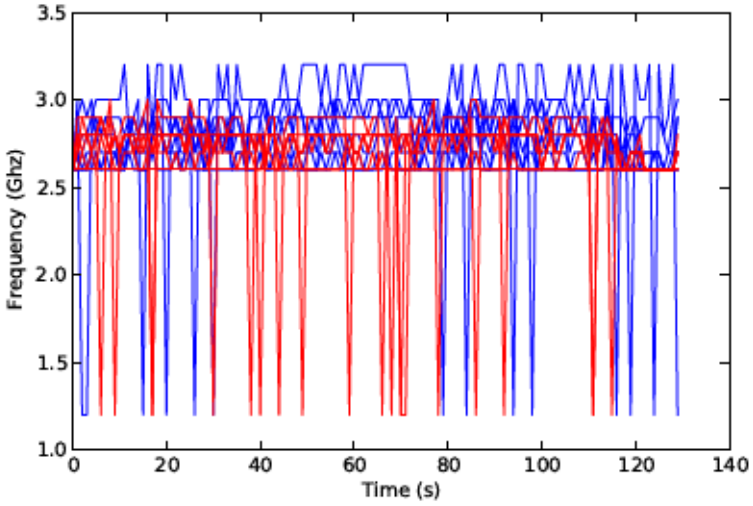


Figure 4.2: *CPU core frequencies during a parallel sequential file write.*

4.2 Strategy for reducing CPU cores thermal imbalance

This section focuses on the reduction of uneven heat accumulation among CPU cores in the system in order to reduce the thermal variation demonstrated in Section 4.1.

4.2.1 Strategy and design

To achieve a reduction in thermal variation across CPU cores, our strategy focuses on dynamically throttling cores based on their temperature in order to reduce temperature variation. Our approach consists of a runtime and kernel software which aims to minimize the temperature deviation between cores by dynamically throttling individual cores up or down.

We propose running a controller algorithm periodically with time interval T. This algorithm adjusts the *throttle ratio* for each CPU core, and proceeds to wait for another T seconds for temperature changes to take effect before running again. The *throttle ratio* is considered to be a value between 0% (no throttling) and 100% (no tasks will be executed). For instance, a throttle ratio of 50% means that, on average, tasks will run for $T/2$ seconds during a time interval T. Initially, all CPU cores start at a throttling ratio of 0%, and throttling up/down occurs in a step-wise fashion in increments or decrements of i ($0\% < i < 100\%$). We consider a target temperature and an acceptance window, to take into account the fact that it would be very difficult to have all CPU cores operating at exactly the same temperature all the time. If the core temperature is close to the target temperature, no action is taken. The core's throttling ratio is increased by i if the temperature is above the acceptance window, and decreased by i when it is below the acceptance window, as depicted in Figure 4.3.

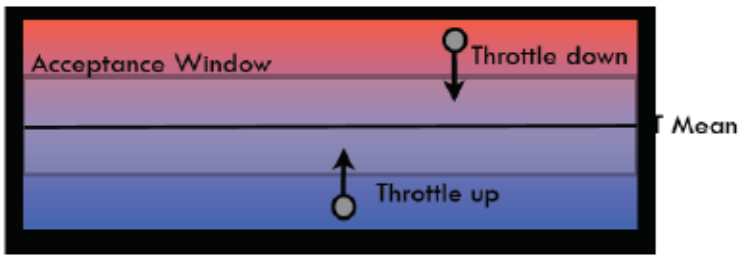


Figure 4.3: *Strategy for reducing temperature deviation among CPU cores.*

Next, we will discuss the details of how we implemented the mechanisms that allow us to throttle individual CPU cores.

4.2.2 Implementation

As was demonstrated in Section 4.1, thermal imbalance occurs on a per-core and per-package fashion. If the goal is to reduce this variation in temperature among cores, a mechanism that allows the operating system or a runtime to throttle individual CPU cores is needed.

Several known mechanisms exist for throttling down CPUs:

- Power capping through Running Average Power Limit (RAPL). RAPL allows the kernel to set a power budget which the CPU will enforce through throttling.
- Software controlled CPU clock modulation (T-States). T-states leverage a clock gating mechanism to reduce the percentage of time the processor is doing useful work.
- Voltage/frequency pair selection (P-States). Dynamic voltage and frequency scaling is a technique employed by processors to provide dynamic performance and power proportionality.
- CPU offlining. This mechanism logically removes CPU cores from the system.
- Idle injection (C-state injection). Idle states put the CPU into an idle power-saving mode.

Of all of these mechanisms, idle injection is known to be the most efficient [LB13, GHBD+09, LWW08]. Idle injection consists in forcing a target CPU core into an efficient idle/sleep state, during which power usage is significantly reduced and no tasks are able to run. Due to its efficiency, idle injection becomes our preferred mechanism for throttling down CPUs.

On x86 processor architectures, idle injection leverages the MONITOR and MWAIT instructions. The MONITOR instruction is used to wait for writes targeted at a specified memory address. MWAIT is used to make the issuing thread enter a power-efficient sleep (C-state). In practice, MONITOR is set up to target

changes to the task's need_resched flag in the Linux kernel process table in order to be woken up whenever the kernel needs to re-schedule the task.

For leveraging this mechanism, we develop our own mechanism for throttling an individual CPU core to a specific throttle ratio. The mechanism we developed works as follows. Our implementation consists of a kernel module, which spawns one high priority kernel thread (placing it on the kernel realtime FIFO queues) per online CPU core. Each kernel thread is pinned to exactly one CPU core. In order to achieve a throttling ratio of $r, 0 \leq r \leq 1$ during a time interval T, a thread will activate a C-state deep sleep for a duration of $T \times r$, and make sure our driver will be woken up by setting a timer interrupt on the same core it is running. For the remaining time, the driver will cause the thread to yield to user threads for a duration of $T \times (1 - r)$, allowing computation to happen.

Our driver sets up kernel timers in order to periodically monitor CPU core temperatures. In addition, average temperature and standard deviation are computed. These data are then fed to a temperature controlling procedure, *ThermClamp*, detailed in Algorithm 3, which will manage per-core throttling ratios for each of the threads that perform idle injection. This results in a feedback control mechanism that reduces temperature variations. Note that since the temperature takes time to converge, we always wait for a certain amount of epochs (one epoch is the time period between *ThermClamp* runs) before taking corrective action on each CPU core.

4.2.3 Evaluation

We evaluated our solution on a two socket system equipped with two identical Ivy Bridge 8-core Xeon E5-2670 and 64GB of DRAM, running Linux 4.2. We experimentally found a wait window of 20s, a 5% step, and an acceptance window of $mean(temperatures) + stdev(temperatures)$ to work well for our purposes. We determined these values by trial and error, and found the combination of wait window and throttle step to be important, as a small wait window or a big throttle step will cause the undesired effect of an oscillating temperature. With these parameters, our technique reduces temperature variation by 50%. Figure 4.4 illustrates a visual comparison of core temperatures with thermal throttling (left) and no throttling (right).

The downsides of this approach are the following. First, idle injection obviously results in performance degradation, which we found to be proportional to the throttling ratio and is highly dependent on the temperature. During this time, the CPU is not able to do actual work, which result in longer runtimes and increased energy consumption. Second, transitions to and from C-states introduce significant performance variations, as depicted in Figure 4.5. We found that there is a necessary trade-off between reducing the temperature variation and the impact on raw performance, where the user would need to pick her desired outcome. In the next section, we focus on a technique that adapts CPU frequency on demand depending on the I/O phase in order to drastically reduce power consumption, without impacting performance.

Algorithm 3 Controller algorithm for reducing temperature variation.

Require: This procedure is called periodically.

`throttleRatio` is initialized for each core to 0.
`adjustAgo` is initialized for each core to 0.
`avgTemp` is the average temperature of all cores.
`acceptWindow` is a threshold for tolerable temperature deviation margin.
`step` is the throttling ratio increment step.
`wait` is the amount of time/epochs to wait between actions to let temperature converge.

procedure THERMCLAMP(avgTemp, acceptWindow, step, wait)
    for each online CPU core i **do**
        $t_i \leftarrow \text{temperature}(core_i)$
        $d_i \leftarrow t_i - avgTemp$
        if $adjustAgo_i \geq wait$ **then**
            if $d_i > acceptWindow$ **then**
                $throttleRatio_i \leftarrow throttleRatio_i + step$
            else if $d_i < -acceptWindow$ **then**
                $throttleRatio_i \leftarrow throttleRatio_i - step$
            end if
            $adjustAgo_i \leftarrow 0$
        else
            $adjustAgo_i \leftarrow adjustAgo_i + 1$
        end if
    end for
end procedure

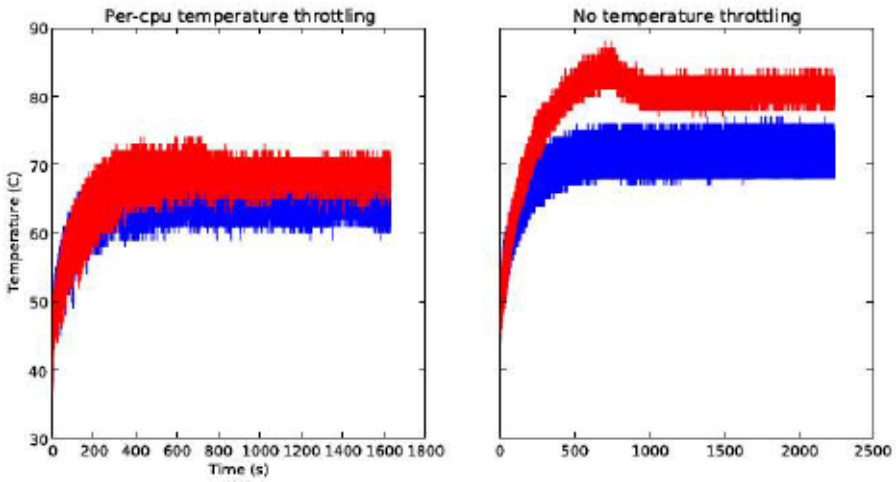


Figure 4.4: *Strategy for reducing temperature deviation among CPU cores.*

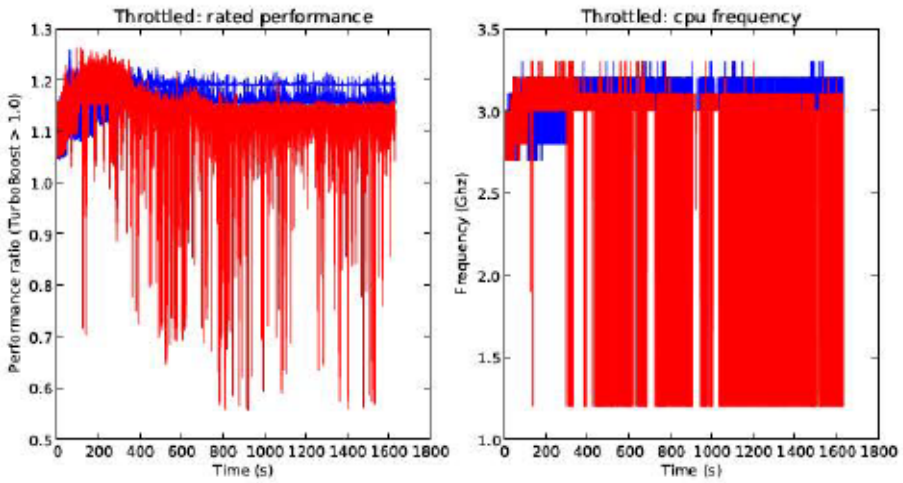


Figure 4.5: *Impact of reducing thermal variation on performance variation. CPU rated performance (left), CPU frequency (right).*

4.3 I/O and thermal-aware thread placement

In this section, we focus on exploiting the CPU-level heterogeneity demonstrated in Section 4.1 to more efficiently manage workloads that combine I/O and computation. More precisely, we evaluate the impact of thermal-aware thread placement. Our hypothesis is that, given two workloads, one that is CPU intensive and another that performs I/O operations, we can improve the efficiency simply by matching specific workloads to cores based on their running temperature.

4.3.1 Strategy and design

We propose a thermal-aware runtime which does intelligent thread placement in order to increase energy efficiency. Our solution places threads on different cores based on thread workload characteristics and core temperature. Thread placement can be based on different thermal-aware policies. We implemented three policies for placing I/O and compute threads in a thermal-aware fashion, as depicted in Figure 4.6.

Interleaved. I/O and compute threads are interleaved through system CPU cores ordered in a cold to hot fashion.

Cold I/O. I/O threads are placed on the colder CPU cores, while compute threads are placed on the hotter cores.

Hot I/O. I/O threads are placed on the hotter CPU cores, while compute threads are placed on the colder cores.

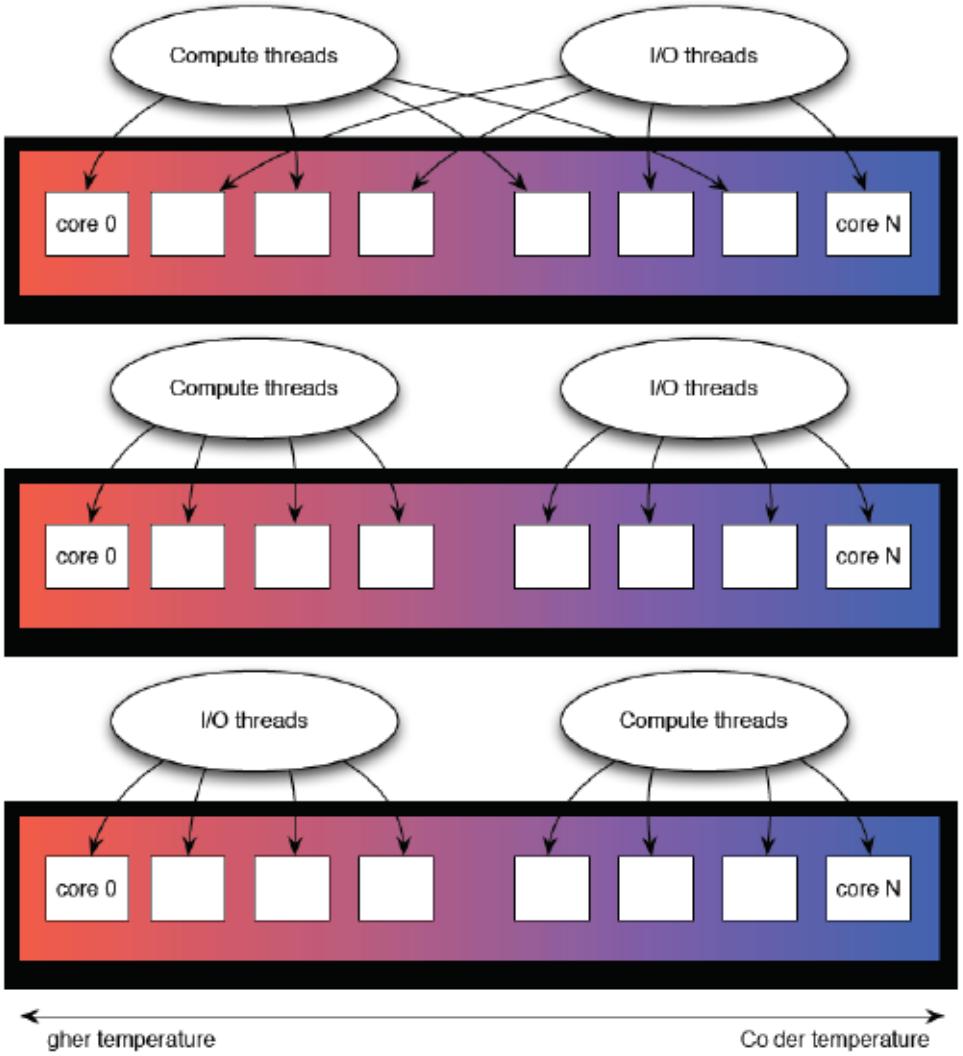


Figure 4.6: *Thermal-aware thread placement policies. Interleaved compute and I/O threads (top), map compute to hottest (middle), map compute to coldest (bottom).*

4.3.2 Evaluation

We evaluated our solution on a two socket system equipped with two identical Ivy Bridge 8-core Xeon E5-2670 and 64GB of DRAM and Linux 4.2. For evaluating these policies, we choose one computational workload that is very CPU intensive, and an I/O-intensive workload. We dedicate half of the cores in the system to run the I/O-intensive workload described in Section 4.1. For stressing the CPU on the remaining half of CPU cores, we do a parallel (one thread per core) run of DGEMM (Double precision GEneral Matrix to Matrix Multiplication) in a loop, leveraging

Intel's Math Kernel Library implementation.

We pin each thread to a different CPU core using different policies, as depicted in Figure 4.6. The first policy (top) interleaves computation and I/O threads between hotter and colder cores. The second policy (middle) assigns all computation to the hottest cores and I/O threads to the coldest cores. The third policy (bottom) assigns all computation to the coldest cores and I/O to the hottest cores.

In our case, hottest and coldest cores were split evenly between both CPU sockets, as evidenced by Figure 4.1 in Section 4.1. We found that the policy that interleaves threads achieves the worst performance and energy efficiency due to the effects of Non-Uniform-Memory-Access (NUMA), as threads stall more often when accessing regions of memory in a non-local node. However, when comparing the other policies where all threads perform DGEMM computations on the same CPU, local memory accesses are not penalized, resulting in more performance. The main difference between both policies was a CPU energy consumption of 3% in favor of the policy that places I/O threads on the colder nodes.

4.4 I/O-aware CPU performance and power management

This section focuses on providing coordination between the software layers responsible for storage I/O and CPU power and performance management. By increasing coordination between the I/O storage and CPU layers, we expect to improve the energy efficiency of I/O workloads.

4.4.1 Strategy and design

The main goal of our strategy is to make storage I/O more energy and power efficient by making the software layer responsible for managing CPU power and performance aware of I/O phases. We achieve this by making the CPU management layer I/O aware and adjusting the CPU frequency and voltage pairs adaptively, depending on the state of I/O operations in the system. We leverage the knowledge about data movement and performance regimes gathered in Chapter 3 in order to provide I/O awareness to the CPU management layer.

The system is considered to be in one of the following three regimes during write I/O activity:

- There is no I/O (Idle).
- Writes are asynchronous and data are being moved to the page cache at a fast pace.
- The operating system's dirty memory control algorithms block writes while the background write-back operations flush the dirty data to disk.

In Section 3.3.2 we were able to identify one more regime, as depicted in Figure 3.3.2. However, to test our hypothesis that I/O-aware CPU management can improve the energy efficiency, we only need to dynamically adjust CPU performance to match I/O regimes that are CPU-demanding. We therefore opt for a simpler solution where we consider the aforementioned three regimes.

We monitor each process's I/O activity as well as the amount of data being dirtied, and manage CPU performance states accordingly. We consider the system to be in one of the two active regimes based on whether dirty memory increases (first regime) or not (second regime). The first regime is more CPU-demanding, while the second regime only requires the CPU to service I/O interrupts. The operating system's CPU performance state controller algorithms are not I/O aware and increase or decrease performance states based on CPU demand and utilization. We observed that the current algorithm keeps the CPU at a power demanding state during I/O. This results in inefficient use of CPU power resources, as power demanding performance states are not required during all I/O phases.

As part of our strategy, we propose a mechanism that dynamically adjusts the performance state of those CPU cores dedicated to I/O based on the current I/O regime. When there is no I/O, the CPU core resumes the traditional strategy of scaling performance states solely based on CPU utilization.

To achieve this, we implement a CPU management driver that is I/O aware and is able to use this information to switch performance states more intelligently. We develop a kernel module that replaces Intel's p-state selection driver. Our implementation is able to detect I/O activity and adapt the CPU frequency and voltage pairs accordingly, while behaving as usual otherwise. Our driver monitors each task's dirty rate activity, and determines in which of the aforementioned regimes each CPU core is in. During I/O activity, we switch the corresponding CPU core to p-state p_{dirty} when we detect the first regime, and to p_{flush} when we detect the second regime. p_{dirty} is a higher demanding performance state than p_{flush}.

However, the selection of the appropriate values for p_{flush} and p_{dirty} will depend on particular properties of each device, and can therefore not be hardcoded to be the maximum and minimum p-states. Depending on the characteristics of system devices such as CPU power efficiency, memory bandwidth, and disk throughput, selecting for instance the lowest p-state might not be beneficial, as it could incur longer runtimes [SHM12]. Therefore, our solution runs a test during system initialization, investigating through every possible combination and measuring the power efficiency. This mechanism automatically configures p_{flush} and p_{dirty} to their optimal values.

4.4.2 Evaluation

The experiments in this chapter have been performed on a two socket system equipped with two identical Ivy Bridge 8-core Xeon E5-2670 and 64GB of DRAM, running Linux 4.2. We evaluate our solution running an I/O-intensive workload that performs a parallel file write on an ext4 file system. We run a process on each of the 16 CPU cores that opens a new file and writes 1GB of data sequentially. This is performed with our I/O-aware solution and with Intel's newest p-state driver. Each

experiment was repeated 5 times in order to obtain average and deviation values for total CPU energy consumption, total runtime, and average CPU core temperature. Figure 4.7 shows that the per-core CPU frequencies on our I/O-aware CPU management solution are much lower, leading to a more energy efficient use of CPU resources.

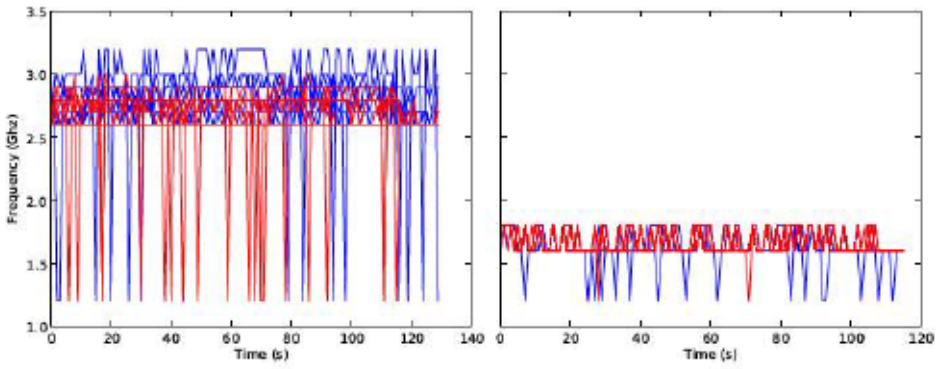


Figure 4.7: *CPU frequencies during write workload using the default CPU management algorithm (left) and our I/O-aware solution (right).*

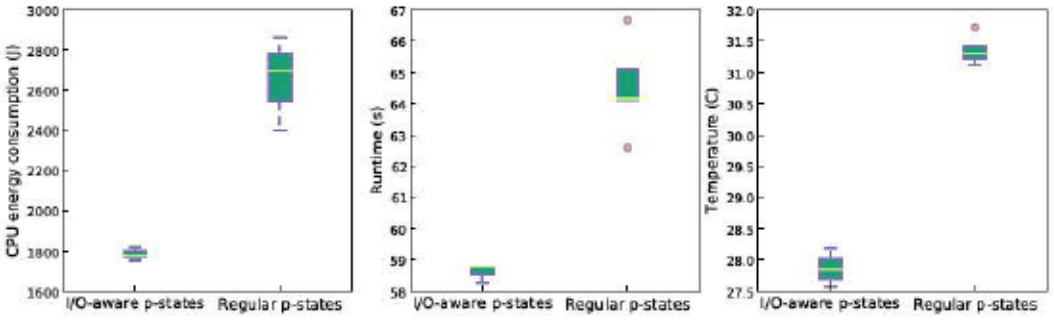


Figure 4.8: *I/O-regime aware p-state selection driver consumes 33% less energy (left) than Intel's driver during write operations, takes 10% less time (middle), and decreases average CPU core temperature by* $3.5°C$ *(right).*

By adaptively setting performance states based on I/O performance regimes, we are able to reduce CPU energy consumption during write I/O by an average of 33%. Figure 4.8 depicts the difference in average CPU consumption between our solution and the Linux default CPU p-state driver. As a result of decreasing stress on CPU performance, our solution also achieves a temperature reduction of $3.5°C$. Surprisingly, we also observed the runtime values to improve as well by an average of 10%. Due to the counter-intuitive nature of this result (we would expect the experiment running on higher frequencies to consume more energy but have equal or lower run times, not longer), we carefully repeated the experiments monitoring per-core frequency, total written data, and paid special attention to uncaught errors,

but in fact found the performance to have improved. In fact, other works have already observed and studied such counter-intuitive outcome [CLGC14], leading us to believe that our results are indeed correct.

In conclusion, we have developed an I/O and regime aware CPU performance state controller that reduces energy consumption and total run time for write I/O operations.

Chapter 5

Virtualized I/O and data sharing

Over the past years, the cloud, especially Infrastructure as a Service (IaaS) offerings, have seen a staggering rise in popularity due to the convenience and the economic pricing models. Even cloud service providers are starting to offer custom-sized HPC clusters as a service, with cloud-like pricing models. This can be of special interest to organizations who can not afford to acquire expensive HPC cluster hardware, or would require a different amount of computing power to meet computing demand for each project. From a user perspective, the cloud offers elastic and scalable computing power. Virtualization is a key ingredient to this advantage. However, HPC applications are often constrained by I/O, a problem which is made worse when running in a virtualized environment.

With scientific computing in the cloud gaining popularity and using every time larger data sets, high performance storage I/O in virtualized environments is substantially increasing in importance. However, exploiting the performance potential of the the storage I/O on today's virtualized architectures is complex, due to the limitations of POSIX standard for storage I/O and the lack of integration of related mechanisms such as data sharing, storage I/O coordination, relaxing the consistency semantics, and data locality awareness.

In this chapter we present a flexible I/O virtualization solution, which captures POSIX I/O library calls and forwards them to the host, achieving two goals. First, system calls are avoided in the virtualized environment. Second, as the host performs I/O calls on behalf of virtual domains, data sharing among them becomes more efficient as less data copies are performed. Our solution is evaluated and compared to similar I/O forwarding technologies.

We extend this idea about efficient data sharing by developing VIDAS (Virtualized DAta Sharing), an object-based virtualized data store that targets to in-

tegrate the above mechanisms through a simple powerful interface. VIDAS can be used to efficiently and consistently share access to externally stored data in virtualized environments based on a shared pool of storage objects. We show how VIDAS can be used for straightforwardly implementing I/O coordination and data sharing for two common high-performance patterns: inter-domain write-reader and inter-domain collective I/O. We present the implementation and evaluation of VIDAS for the Xen virtualization solution. Even though implemented in Xen, VIDAS interface is independent of any virtualization solution.

The remainder of this chapter has the following structure. Section 5.1 presents our virtualized I/O forwarding solution. Section 5.5 introduces the novel abstractions and mechanisms for data sharing in virtualized environments, VIDAS. Section 5.6 presents the VIDAS implementation. Section 5.7 describes two use cases for collective I/O, read-write file sharing and collective I/O. Section 5.8 evaluates our solution for several collective I/O patterns between virtual machines. Finally, Section 5.9 summarizes the contributions of this chapter.

5.1 I/O virtualization architecture

Our proposed I/O virtualization and framework architecture is shown in Figure 5.1.

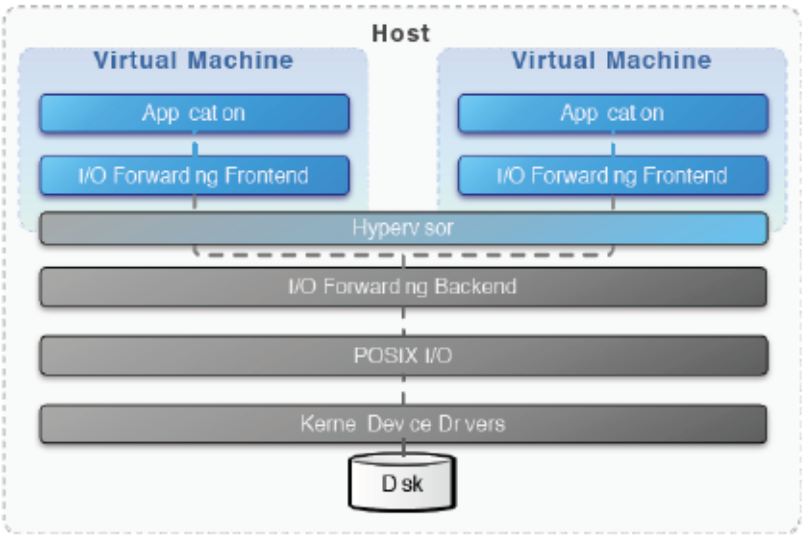


Figure 5.1: *Comparison of the BT class B benchmark read and write times. Less is better.*

Our solution relies on the GVirtuS framework that provides a flexible mechanism for virtualizing I/O. One of the main features of GVirtuS is its modularity, allowing the development of split-drivers in a straightforward fashion by requiring implementation of only stub routines in the front-end and service routines in the back-end. The GVirtuS system consists of a front-end which runs in the virtual machine and a back-end service running in the host.

While this type of general-purpose virtualization framework has been previously demonstrated to run in high-performance computing scenarios where GPU virtual-

ization is needed [GML+11, GMAC10], we propose the addition of a distributed back-end architecture and provides an I/O virtualization evaluation. The system is highly flexible, allowing different methods of communication between applications and back-end through a generic interface. Communication can be done through shared memory, UNIX sockets and TCP sockets.

For communicating a front-end and a back-end we employ a technique called I/O forwarding. In its original form, I/O forwarding addresses the specialization of compute and I/O in supercomputer architectures by shipping I/O calls from compute nodes to dedicated I/O nodes [ACI+09]. The I/O forwarding in typically implemented as Remote Procedure Calls. Usually, I/O nodes perform I/O on behalf of the compute nodes. In our solution, instead of calling the VM-local I/O library functions, applications have their library calls transparently forwarded to the front-end, which in turn communicates the call to the back-end through a high-performance communication channel. This architecture enables a high degree of flexibility, as libraries which override local library functions can be developed and integrated into a working system without recompiling target applications.

The virtualized I/O solution captures POSIX library calls and forwards them in order to avoid running I/O system calls in the virtualized environment. In a similar fashion to lightweight RPC calls, function and parameters are marshalled and sent to the host through the ringbuffer mechanism implemented over shared memory, as depicted in Figure 5.2.

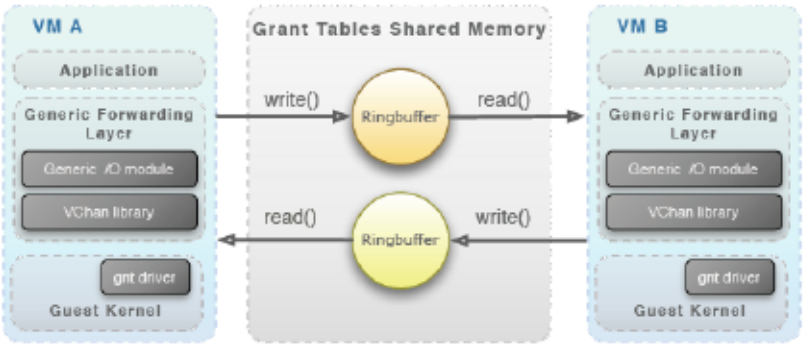


Figure 5.2: *Forwarding of I/O operations between virtualization domains is achieved with low overhead through the use of inter-domain shared ring buffers.*

In the simplest mode, each virtual machine will execute a POSIX I/O call which will then be forwarded to the back-end running in the host where they are run in a safe, isolated environment. In order to achieve isolation, the open() C library function ensures that virtualized I/O operations are not able to access the host file system by checking the pathname parameter and limiting access to a jailed folder.

5.2 I/O virtualization evaluation

The computer system used for the evaluation is equipped with an AMD Opteron(tm) Processor 6168 (two six-core processors at 1.9 GHz), 32 GBytes of RAM, and a Barracuda ES.2 SATA 3.0-Gb/s 1-TB hard drive. The chosen VM hypervisor is Xen version 4.1, with dom0 and domU running kernel version 3.0.0. The MPI distribution was MPICH2 1.4.1. We ran all tests with one process per VM node. The host and file systems are connect through a Gigabit ethernet network.

5.2.1 Concurrent file write

We have evaluated our solution by testing the scalability of concurrent file writes. We evaluate the performance of file I/O when increasing the number of concurrent VMs performing I/O. To achieve this, we run a synthetic benchmark which writes a 512MB file sequentially to disk using a 64KB block size. Each write process runs on a different VM, and we evaluated this benchmark running on 1, 2, 4, and 8 VMs using regular file writes and our I/O forwarding solution. The results are shown in Figure 5.3. As we scale the number of VMs and concurrent writers up, performance drops significantly when running on vanilla Xen, while our solution achieves the full performance allowed by the host. Our solution takes advantage of I/O forwarding, which ultimately runs write() operations on the host. Therefore, data are able to be better coordinated by the host's I/O stack, reaching the full I/O bandwidth even when running 8 concurrent writers. Without I/O forwarding, write() operations are handled by each VM's operating system's I/O stack. The host ends up receiving uncoordinated block device requests from each VM separately and is unable to optimize write access to be more sequential, which explains the performance drop.

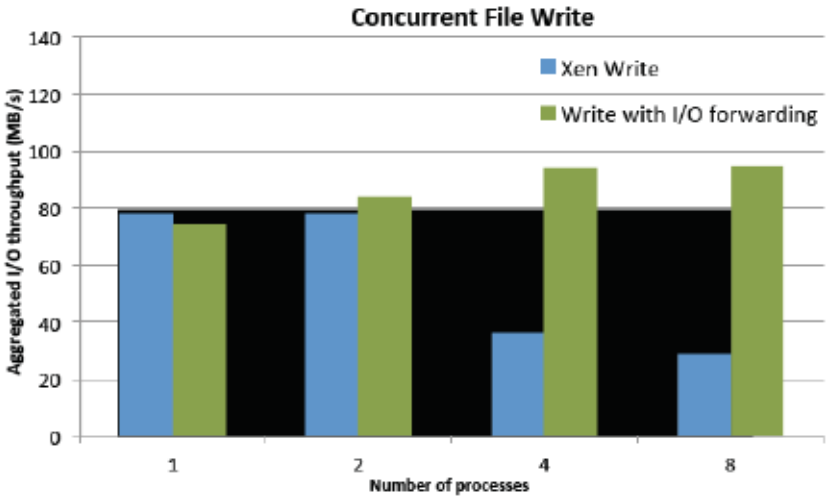


Figure 5.3: *Comparison of concurrent VM writers with and without I/O forwarding. Less is better.*

5.2.2 BTIO benchmark

We have evaluated our solution using NASA's BTIO benchmark, which solves the block-tridiagonal (BT) problem. Initially, all compute nodes collectively open a file and declare views on the relevant file regions. After every five computing steps, the compute nodes write the solution to a file through a collective operation. At the end, the resulting file is collectively read, and the solution is verified for correctness. We evaluate our solution using 1 to 16 processes and a data set class B, resulting in a file of 1697.93 MBytes.

We have evaluated five different mechanisms: an I/O forwarding solution for supercomputers (IOFSL), our proposed solution (IOGVirtuS), guest virtual nodes accessing data through a local NFS (Guest + NFS), guest virtual nodes acceding data through an external OrangeFS filesystem (Guest + OrangeFS), and a non-virtualized solution in which MPI processes are deployed in the same physical node (Host). IOFSL leverages the ZOID project at Argonne National Laboratory to perform I/O forwarding to dedicated I/O nodes in a similar fashion to Blue Gene systems [ACI+09]. Results are shown in Figure 5.4. The results depict that Guest+OrangeFS obtained the worst times due to the extra network overhead. Guest+NFS obtained the best results for one process, however the trend shows that this solution offers a small grade of scalability due to the file locking mechanism. Our solution obtained generally better results than IOFSL, but results show that there is still big room for optimization, which will be achieved after proper instrumentalization. However, our solution opens the door to better scalability and power-efficiency improvements.

Figure 5.4: *Comparison of the BT class B benchmark read and write times. Less is better.*

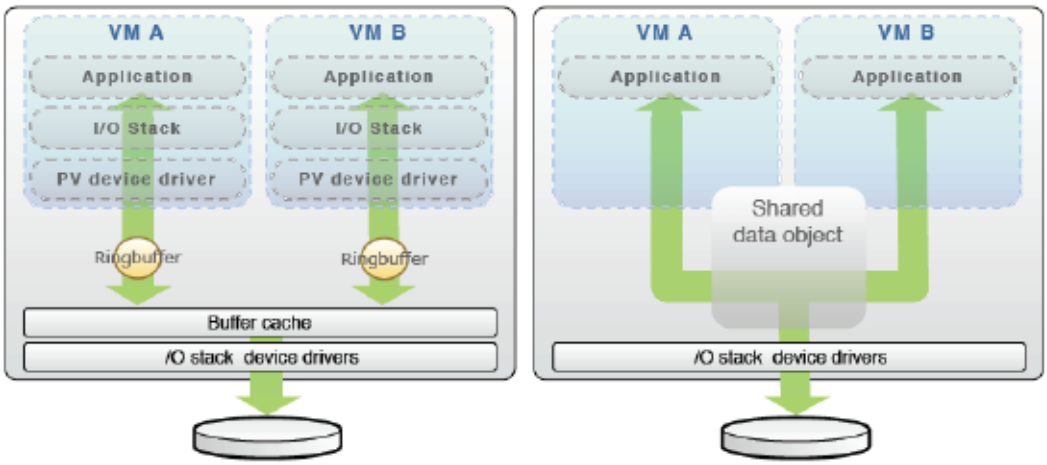
5.3 Efficient virtualized data sharing

So far we have presented a storage virtualization solution which is based on I/O forwarding, extending the GVirtuS virtualization framework. In addition we evaluated our solution and compared it with native host performance and different I/O virtualization solutions such as IOFSL and other network file sharing technologies. Our system shows close performance compared to other solutions, and in some cases noticeable performance improvements where the host is able to coordinate file storage access. However, results show that there is still room for improvement.

As a result, the need for an efficient virtualized data sharing solution which optimizes for coordinated I/O access and minimizes data movement arises. Inefficient data movement happens due to the amount of data copying required for communicating through the use of ring buffers from each domain to the host and among domains. However, I/O forwarding makes extensive use of ring buffers in order to transfer all data between domain and host. Ideally, a data sharing solution would be capable of offering more efficient data sharing mechanisms and novel semantics for coordinating collective I/O. We hypothesize that such a solution would improve performance and reduce energy consumption.

Therefore, we propose a novel solution that focuses on reducing data movement in virtualized environments and provides flexible I/O semantics for efficient and coordinated storage I/O. VIDAS is our solution for efficient virtualized data sharing among domains. When performing collective I/O operations, instead of requiring data to be copied to the host and then perform one additional copy to every other domain (or even multiple copies per domain, if the use of ring buffers is employed), VIDAS allows a different data flow which minimizes data copies. Figure 5.5 compares how data flows on traditional solutions versus our proposed solution. Note that our solution supports the capability of collectively sharing data among an arbitrary number of domains (not just two as depicted), minimizing data copies.

However, introducing these concepts creates the need for new abstractions and semantics in order to efficiently share data using shared data objects. Also, it needs to be clarified where such a data sharing solution fits in the I/O stack. These issues will be detailed next.



(a) *Others dataflow.* (b) *VIDAS dataflow.*

Figure 5.5: *Comparison of VIDAS data flow to other solutions.*

5.4 The role of VIDAS as a storage virtualization solution.

We consider VIDAS as a paravirtualized object-level storage pool. As such, it can not be straightforwardly categorized in the classification presented in Section 2.7. We see VIDAS as an intermediary layer between disk-level virtualization and file system-level virtualization. On one hand, VIDAS can be used to consistently share access to a non-shared disk. On the other hand, VIDAS can serve as an intermediary layer for building a shared paravirtualized file system by allowing a straightforward implementation of a shared name space or a shared buffer cache on top of storage objects. In general, VIDAS can be used for offering guests high-performance data sharing and locality awareness based on a shared pool of storage objects. The shared pool consists of objects that can be mapped to every virtual machine running on the same host. Our memory sharing mechanism differs from existing mechanisms for communicating and data sharing between virtual machines such as XenSocket [ZMRG07], XWAY [KKJ+08], and others [LJSL09, HKGP07], which typically share a page between two domains.

5.5 Abstractions and mechanisms for virtualized data sharing

This section introduces the novel abstractions and mechanisms for data sharing in virtualized environments. Our virtualized data sharing solution is based on two abstractions: containers and storage objects (as shown in Figure 5.6). A *container* (described in Section 5.5.1) is an abstraction which allows data sharing between virtual domains running on the same physical node. The data sharing can be done at the granularity of a data *storage object* (described in Section 5.5.2). All domains

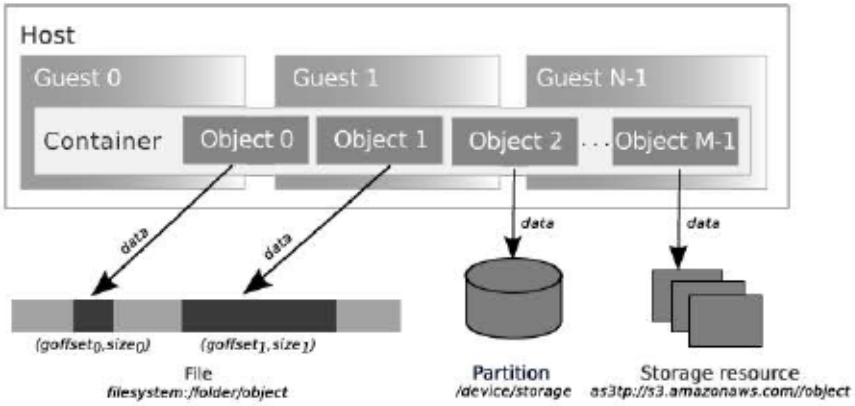


Figure 5.6: *Several guests sharing objects through a container. Each object is mapped on an external storage resource.*

Table 5.1: *List of container operations.*

Container operations
int container_create(char *name, int domain_ids[])
int container_destroy(char* name)
int container_attach(char *name)
int container_leave(char* name)

with access to a common container can use it to share data objects among each other. Storage I/O coordination, data sharing, asynchronous I/O can be done at object-level.

5.5.1 Containers

A container in VIDAS is an abstraction which facilitates storage object sharing across virtual domains. A container has a unique name, which has to be known by the domains who want to share it. VIDAS provides a restricted set of container operations as listed in Table 5.1.

A container is created by an initiator domain by specifying a unique name and an array of potential domains that are allowed to share it. A container can be destroyed by the initiator domain only if there is no other domain sharing it. A domain different from the initiator can share a container by calling `container_attach` and unshare it by calling `container_leave`. After sharing a container, domains are able to start to manipulate and access storage objects shared by any one of them as described in the next section.

5.5.2 Storage objects

A storage object in VIDAS is an abstraction for data sharing across domains. Each storage object is uniquely associated to an external storage resource through its

Table 5.2: *List of object operations.*

Object metadata operations
obj_handle_t object_create(char* ext_storage_rsc, size_t offset, size_t size, char* cname)
obj_handle_t object_join(char* ext_storage_rsc, size_t offset, size_t size, char* cname)
int object_get_locality(char* ext_storage_rsc, obj_handle_t *objects[])
int object_leave(obj_handle_t o)
int object_destroy(obj_handle_t o)
int object_getattr(obj_handle_t o, char *name, void *value, size_t size)
int object_setattr(obj_handle_t o, char *name, void *value, size_t size)
Object data access operations
int obj_write(obj_handle_t o, char *buf, size_t offset, size_t sz)
int obj_read(obj_handle_t o, char *buf, size_t offset, size_t sz)
int obj_flush(obj_handle_t o)
int obj_update(obj_handle_t o)
Object synchronization operations
int object_wait(obj_handle_t o, char **bufp)
int object_notify(obj_handle_t o, char *buf)

name. The external storage resource can be a file from a file system, an object from a storage system, a disk partition, or any other storage resource that can be uniquely identified through a name and offers a linear address space. For instance, the external storage resource can be a file from a locally mounted NFS or a URL of a remote object stored in Amazon S3. We assume that a simple get/put interface is available for accessing these external storage resources and we will concentrate on the node-level data sharing in a virtualized environment.

The storage objects are different from traditional POSIX files in several aspects:

- Each object is associated with a user-extensible list of name-value attributes.
- Strong consistency is not enforced, but optional.
- Data writes to external storage resources can be guided by a configurable policy such as write-through or write-back.
- Applications can learn if object data is cached in memory, providing locality awareness.
- Operations on objects are stateless.

In the remainder of this section we discuss storage object operations (shown in Table 5.2).

Metadata operations

Each VIDA storage object can be uniquely associated to an external storage resource through the `obj_create` call. After creation, any other domain can share the object by joining through the common container (`obj_join`). The `obj_create` operation associates a data object to a container, i.e. reserves shared memory in a container for the given data range of an object. Domain access control is enforced through the container associated to the object, which specifies in which domains the object is accessible. Object attributes can be manipulated using the `object_getattr` and `object_setattr` operations. Predefined attributes include: "name" (the external storage resource name, i.e. a file or an URL), "data" (a pointer to the object's data), "offset" (the offset where the object maps on to the storage destination), "size" (size of the region of the data item shared by this object), "container" (container through which the object is shared), "synchronized" (if "true", the operations on this object are atomic), "write policy" and "read policy" (which are further discussed in the next subsection). Further attributes can be defined by users through `object_setattr`.

Data operations

After an object has been shared by a domain (either through `obj_create` or `obj_join`), it can be accessed from the domain through standard `obj_read` / `obj_write` calls. Alternatively, the pointer to the object data can be retrieved through an `object_getattr` operation on the "data" attribute and, subsequently, directly accessed. The first alternative offers "opaque" access to the object and can be used together with the attribute "synchronized" in order to provide atomic access to the object. In the atomic mode (the attribute "synchronized" has the value "true"), accesses to non-overlapping object intervals can proceed concurrently. The second alternative provides a "zero-copy" mode, for which object modifications from any domains become instantaneously visible to all the other domains. However, in this mode the user is responsible to enforce access consistency.

The "update policy" and "write policy" object attributes reflect how the data flows between a VIDAS object and the represented external storage resource. The "update policy" decides if external object updates are applied "lazy" or "eager". For lazy updates the object is updated from the external storage resource, only when the user calls `object_update`. The "write policy" refers to how object modifications are propagated to the external storage resource. If the write policy is "write-through", object modifications are propagated right away. For the "write-back policy", the object modifications are propagated only when the user calls `object_flush`.

Synchronization operations

As discussed in the previous subsection, atomic access to an object is provided when the "synchronized" attribute is set to "true" and the `obj_read` / `obj_write` are used. For coordinating the access to objects from different domains, VIDAS provides a wait/notify mechanism. A notify operation `obj_notify` sends a message to an object. A different domain can block waiting to receive the message for the

given object by calling `obj_wait`. Section 5.7 shows an example of using wait/notify for implementing a collective I/O write operation.

5.6 Implementation

We have implemented VIDAS interface based on Xen virtualization solution [PFH+05].This section describes Xen implementation details. In order to simplify the reading, we start by presenting the Xen inter-domain mechanisms leveraged by our implementation. Subsequently, we present the implementation of the multi-domain data sharing mechanism. Finally, we discuss container and object implementations.

5.6.1 Xen inter-domain mechanisms

VIDAS implementation leverages three main inter-domain Xen mechanisms: XenStore, shared memory, and ring buffers [PFH+05].

XenStore. XenStore is a key-value centralized data base, which is shared by all Xen domains and is typically used for passing configuration parameters across domains.

Xen shared memory. The Xen mechanism for memory sharing is based on a mechanism called "grant table". Each domain has its own grant table that is shared with Xen. Each entry of a grant table informs Xen about the pages shared by the owning domain with other domains. Each grant table is indexed by a grant reference. A domain dom_i performs the following actions in order to share pages with dom_j. First, it calls a function (`grant_foreign_access`) with two parameters: the target domain dom_j and the number of pages to share with dom_j. This function allocates memory, assigns it to a grant table entry, and returns a grant reference and a local index. The grant reference is passed over XenStore to dom_j, while the local index is used by *mmap* to map the pages to the virtual memory space of dom_i. On its turn, dom_j retrieves the reference grant from XenStore, calls an ioctl function using the grant reference in order to retrieve the local index, and, finally, uses the local index to map the pages to its virtual memory space through a *mmap* call. At this moment the pages are shared between domains dom_i and dom_j.

Xen ring buffers. For inter-domain communication, a producer/consumer circular queue known as a ring buffer, is implemented on top of a shared memory buffer. This ring buffer acts as the transport mechanism between domains for implementing inter-domain communication.

5.6.2 Xen inter-domain mechanisms in VIDAS

VIDAS implementation leverages shared memory for sharing object data, attributes, and other opaque metadata across used domains and ring buffers for communication between the user domains and the host. Figure 5.7 depicts an overview of our implementation, which shows how VIDAS combines ring buffers and shared memory.

The memory is shared between two domains based on the procedure described

in the previous subsection. For sharing a page among n domains, in our solution the domain initiating the sharing inserts in its grant table $n-1$ entries, all of which are associated with the same page. Subsequently, each of the other $n-1$ domains receives a different grant reference, representing the same physical page. The page is then mapped as in a two page case[1].

The ring buffers are used for implementing a lightweight Remote Procedure Call (RPC) based on a front-end, running on the calling domain, and a back-end running on the host. The back-end waits on the ring buffer to receive call messages from the front-end, performs the call, and returns the result in the ring buffer. After performing the call, the front-end waits for the result from the back-end. In this work both the back-end and front-end use polling, while an interrupt based approach was left for the future work. Our initial choice was based on the conclusion of a study, which showed that the use of polling in optimizing the storage virtualization can substantially reduce the overhead of interrupt handling [BYFR+12].

5.6.3 Container implementation

The container management is performed at the host. All container operations are implemented as lightweight RPCs, as described in the previous subsection. When creating a container, the guest forwards the call to the host, which stores the domain IDs sharing access. A subsequent container attach operation is successful only if the calling domain belongs to the list specified by the creating domain. Destroying a container and leaving from a container are simple RPCs that remove or update the container metadata from the host.

5.6.4 Object implementation

Object management is also performed at the host. However, most operations on object data and attributes do not involve the host, as described below.

Seven operations from VIDAS interface rely on RPCs to the host. In order to create an object (`object_create`), a domain follows the steps described above: allocates memory for object data, object attributes and other object metadata, is assigned a grant reference, passes the grant reference to all memory sharing domains, and maps the page to its virtual memory. Finally, it contacts the host with an RPC and informs about the newly created object. At this point each sharing remote domain is entitled to share the object by calling `object_attach`, which maps the object memory to the remote domain virtual memory and informs the host through an RPC. The operations `object_destroy` and `object_leave` undo the create and attach operations, respectively, and inform the host to remove the object from the index. The function `object_get_locality` is implemented as an RPC which returns from the host all the local objects (created or attached) associated to the external storage resource. The other two operations, whose implementation leverages RPCs to the host are `object_flush` and `object_update`, which simply ask the host to flush/update the object to/from the remote storage resource.

[1]We provide this new Xen feature at https://github.com/pllopis/linux/commit/fb6dca

All the other six VIDAS operations rely on shared memory and do not directly involve the host. As the object attributes are stored in main memory, the functions `object_getattr` and `object_setattr` directy work on the shared memory, after implicitly taking a mutex in order to ensure consistency. The mutex is implemented by using Linux atomic test-and-set operations [2]. The data access operations `object_write` and `object_read` access the shared memory directly and are atomic only if the *synchronized* object attribute is set to "*true*". In VIDAS, accesses to an object are serialized only if the accessed domains overlap. For addressing this issue in our current implementation, the object metadata includes an array of mutexes which provide mutual exclusion to overlapping access domains. A further extension to this implementation for providing multiple-reader one-writer access is straightforward.

Finally, `object_notify` and `object_wait` work as well directly on a buffer shared across the domains when object is created/attached. The `object_notify` operation copies the message to the shared memory and atomically modifies a producer pointer, while `object_wait` retrieves the message when available and passes it to the calling domain. We use polling for `object_wait`, while leaving a blocking version for future work and evaluation.

While there are numerous ways to design cooperative data sharing, we made an effort to minimize host intervention where it was not strictly needed. This is an important design decision because context switches result in hypervisor *exit* operations, which are known to be the main cause of performance overheads for virtualized I/O-intensive workloads [GAH+12]. Therefore, operations which manipulate object data and metadata are carried out with minimal context switches. For other operations such as object creation and removal (which are still very fast, as demonstrated in the evaluation), we chose to sacrifice a context switch for consistency and simplicity, by having a single copy of container and object indexes maintained by the host.

[2] http://old-list-archives.xen.org/xen-devel/2009-03/msg01823.html

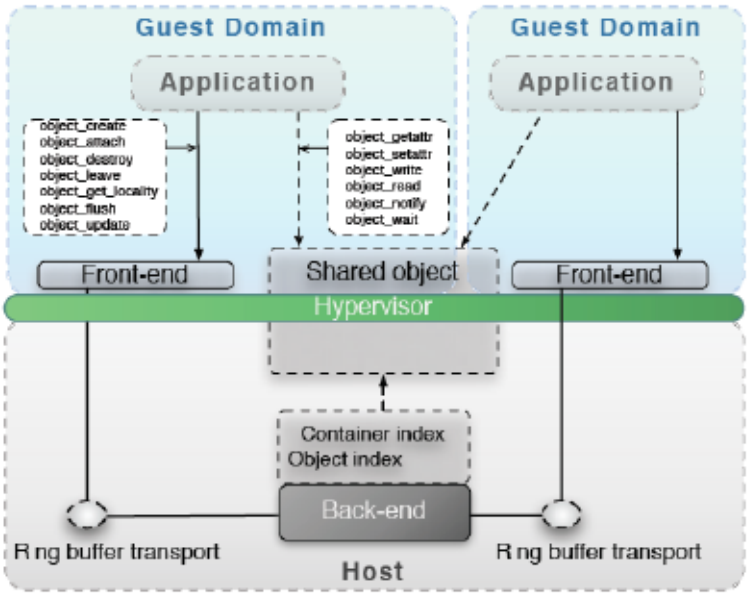


Figure 5.7: *VIDAS implementation. Seven operations rely on RPCs to the host implemented with Xen ring buffers. All the other six VIDAS operations rely on shared memory and do not directly involve the host.*

5.7 Use cases

In this section we present the implementation of two use cases based on VIDAS interface and abstractions: inter-domain write-read sharing of a file (Subsection 5.7.1) and inter-domain collective I/O (Subsection 5.7.2). These two use cases are part of the evaluation in Section 5.8.

5.7.1 Inter-domain write and read sharing

A significant class of scientific applications is based on workflows, in which the output of one process is the input of the next [VAKC+12]. While processes communicating through files within the same host can take advantage of data locality through the buffer cache, processes running within the same host but on different domains require efficient inter-domain data sharing solutions. In this section we show how the building block of a write-read pattern of a workflow can be simply and efficiently implemented based on VIDAS.

Listing ?? shows the implementation. We assume that the writer executes in domain i and the reader in domain j. Before calling the I/O storage functions write or read, domain i creates a container "con" and permits domain j to share it, while domain j joins this container. Subsequently, domain i creates an object of $size$ bytes mapped to the offset $offset$ of the external storage resource $ext\_storage\_rsc$ (a file or a URL), sets the attribute *synchronized* to "true" (in order to enable atomic accesses), writes the data, and notifies the object modification. Domain j joins the object, waits for a notification, and reads the data. In the listing the reading of

```
Write operation executed in domain i

// Share the object with domain j
int domain_ids[]={j};
// Create the container
container_create(''con'', domain_ids);

write(char *ext_storage_rsc, char *buf, size_t offset, size_t size) {
  // Create the storage object
  object_handler_t o = object_create(ext_storage_rsc, offset, size, ''con'');
  // Indicate the all accesses will be atomic
  object_setattr(obj, ''synchronized'', ''true'', 5);
      // Write the data to objects in segments of size b
  object_write(o, buf, 0, size);
  // Notify that the data is available
  object_notify(o, NULL);
  // Destroy the object
  object_destroy(o);
}
// Destroy the container
container_destroy(''con'');


Read operation executed in domain j

// Attach to an existing container
container_attach(''con'');

read (char* ext_storage_rsc, char *buf, size_t offset, size_t size) {
  // Join the object to mapping to a common external storage resource
  object_handler_t o = object_join(ext_storage_rsc, offset, size, ''con'');
  // Wait for the data to become available
  object_wait(o, NULL);
  // Read the data to objects in segments of size b
  object_read(o, buf, 0, size);
  // Unmap the object
  object_leave(o);
}
// Leave the container
container_leave(''con'')
```

Listing 5.1: *Inter-domain write-read data sharing.*

the data is done through the explicit *obj_read* operation. However, it is possible to avoid copying the data involved in this operation, by simply retrieving the pointer to the shared buffer through the "data" attribute and directly use the data.

The way the data modifications propagate from the shared object to the external storage resource can be controlled through the `write policy` attribute (not shown in Listing ??).

The write/read operations presented above can be straightforwardly used as a building block for a producer-consumer implementation or for a data streaming implementation.

5.7.2 Inter-domain collective I/O

I/O intensive applications often face the problem of accessing non-contiguous portions of data. In scientific applications it is often the case that while different processes access non-contiguous portions of data, requests of a group of processes may together span a contiguous portion [TGL99]. The optimizations merging different requests from cooperating processes into a single large I/O operation are referred to as collective I/O [TGL99].

In a purely virtualized environment, efficient collective I/O is difficult to achieve because domains are isolated from each other and data has to be shared through a network file system or network communication protocols. However, VIDAS abstractions and mechanisms allow for efficiently sharing data and coordinating accesses, as shown in Listing **??**. In this case study domains d_0, d_1, ..., and d_{n-1} write non-contiguous pieces of data of sizes given by the $sizes$ vector from the buffers $bufs$ to the external storage resource $ext\_storage\_rsc$ at several offsets given by the $offsets$ vector. We assume that the container “con" has been already created and shared and that the `write policy` attribute is set to “back", i.e. write back. The implementation uses a function get_domain that returns a unique domain name (in Xen we implemented this function by simply requesting this value from XenStore).

Domain d_0 creates an object and all the other domains are sharing it. Subsequently, all domains write the data to the shared object through simple memory copy operations. Subsequently, all n domains notify the object that they have performed the modifications. Domain d_0 waits for all notifications to arrive (for simplicity we show as well the notification sent by d_0 to itself, but this can be obviously optimized away) and, subsequently, flushes the modifications to the external storage resource. Collective read operations can be implemented in a similar fashion.

```
Write is called from domains d_0, d_1, ..., d_{n-1}
We assume domain d_0 will be the aggregator

write (char* ext_storage_rsc, char *bufs[], size_t offsets[], size_t sizes[])
    {
  object_handler_t o;
  // Get my domain and create/share the object
  int my_domain = get_domain();
  if (my_domain == d_0)
    o = object_create(ext_storage_rsc, offset, size, ''con'');
  else
    o = object_join(ext_storage_rsc, offset, size, ''con'');
      // Each domain writes data to the object by calling
      // several times object_write on the object o
  ...
  // Each domain notifies the modifications
  object_notify(o, NULL);
  // Domain d_0 waits for all notifications before flushing the data
  if (my_domain == d_0) {
    for (i=0;i<n;i++)
      object_wait(o, NULL);
    object_flush (o);
  }
}
```

Listing 5.2: *Inter-domain collective I/O write implemetation.*

5.8 Evaluation

We evaluated the VIDAS prototype on a 2-socket system with two identical 8-core Haswell Xeon E5-2630 v3 processors and a MegaRAID SAS-3 3108 storage array with four 1TB disks in a RAID-5 configuration. Each hard disk is a Hitachi HUA722010CLA330 with a capacity of 1TB, a speed of 7200 rpm, and a 32MB cache. Xen version 4.2 runs Linux 3.5.7 as Dom0, modified to support sharing memory pages across an arbitrary number of domains. We also evaluated energy efficiency by obtaining package energy consumption from the processor's Running Average Power Limit (RAPL) interface. Preceding each experiment, we cleared caches, directory entries and inodes from memory using the Linux drop_caches interface. First we evaluate object operations. Then we evaluate the effectiveness of our multi-domain memory sharing mechanism by comparing in-memory inter-domain data sharing with existing alternatives such as MPI and Xen ringbuffers. We compare several synthetic I/O benchmarks with MPI versions, focusing on the performance and energy consumption.

5.8.1 Object operations

In this section we present an evaluation of VIDAS object and container operations. Table 5.3 shows basic container operations execution time. The container operations take between 64μs and 70μs, which correspond to the time of the RPC between the guest and host domains. The creation of an object of one 4096 byte page takes 207μs, representing the time to allocate memory, send the grant reference to the remote

domain, map the page, and perform an RPC for registering the domain. Joining an object takes $98\mu s$, corresponding to the time to map the page and register at the host through a RPC. Leaving an object involves an unmap operation and an RPC and takes $82\mu s$. The other operations involve shared memory and take $2\mu s$ (setting and getting an attribute of 1 byte) and $6\mu s$ (notify/wait a message of 64 bytes).

Table 5.3: *VIDAS operations execution time.*

Container op	**T (μs)**	**Object op**	**T (μs)**
container_create	66	object_create	207
container_attach	64	object_join	82
container_destroy	70	object_notify/wait	6
container_leave	70	object_leave	64
		object_getattr	2
		object_setattr	2
		object_destroy	70

5.8.2 Inter-domain communication

Due to the fact that the goal of this work is to improve data sharing in virtualized environments, we first evaluate data communication between virtual machines without performing I/O. We evaluate broadcasting 128MB data objects to 2, 4, 8 and 16 virtual machines using existing inter-domain memory sharing solutions such as Xen ringbuffers, the OSU broadcast benchmark, MPI broadcast, and our solution. Figure 5.8 shows the effectiveness of our multi-domain memory sharing mechanism. Because the other inter-domain communicators are restricted by Xen's current memory sharing mechanism, which limits the amount of domains sharing a page to 2, they are required to do additional memory copies. Our solution requires a single memory copy and scales better. For fairness, we introduced an additional memory copy per virtual machine in order to obtain different copies of the data object, as opposed to just one shared copy. While this is not required, we felt that it would be more representative due to the fact that each virtual machine is then able to modify its own copy without affecting the others. This is also the reason why performance drops slightly when scaling the number of virtual machines.

5.8.3 Write and read sharing

In this subsection we compare the writer-reader implemented in VIDAS (described in 5.7.1) with a writer-reader based on PVFS2 and NFS using 512 MB files. VIDAS semantics enables us to perform an asynchronous write to disk while readers read object data directly from memory, thus obtaining memory read speeds, while NFS and PVFS2 rely on the disk write buffers for performing reads. We obtained a sustained throughput of over 500MB/s for VIDAS, while NFS and PVFS2 performed at close to 140MB/s.

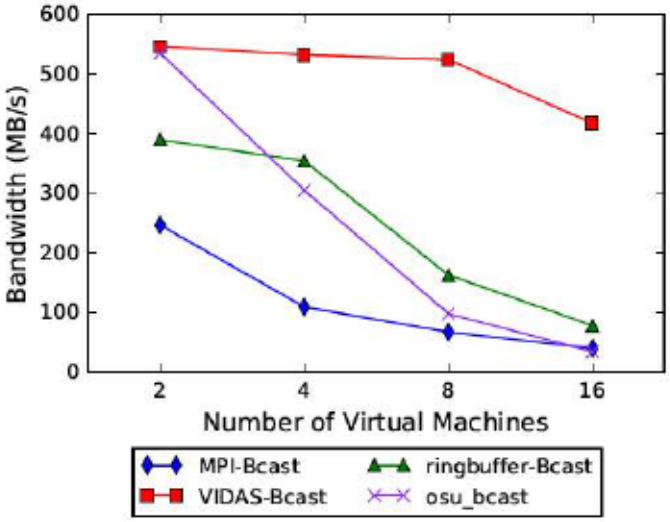


Figure 5.8: *Evaluation of broadcast communication in VIDAS, MPI, OSU benchmark, and Xen ringbuffers.*

5.8.4 Independent shared file read

We also evaluated shared file read, scaling the number of domains which perform overlapping reads of a 512MB file concurrently. VIDAS uses the broadcasting mechanism evaluated in Section 5.8.2. Figure 5.9 shows that VIDAS outperforms PVFS2 and NFS. VIDAS reads data into a shared object and does not require additional memory copies to transfer the data to every other domain. However, for fairness we performed an additional copy in order for each domain to have a different copy. Without this additional copy, the throughput corresponding to a disk file read would have been sustained for any number of virtual machines. We note that VIDAS consistently outperforms ROMIO on shared file systems by a large margin, achieving greater performance (112MB/s vs 1484MB/s for 32 domains) and lower energy consumption (76% less package energy).

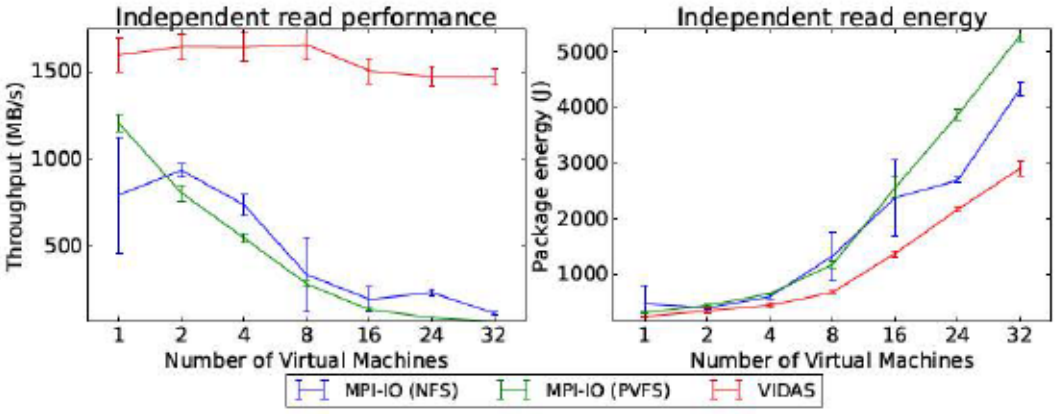


Figure 5.9: *Evaluation of multiple reader in VIDAS, NFS, and PVFS2.*

5.8.5 Collective I/O

In this section we compare the VIDAS inter-domain collective I/O operations (described in Subsection 5.7.2) with a standard collective I/O implementation from ROMIO, the most popular MPI-IO distribution. The collective I/O implementation of ROMIO is based on two-phase I/O [TGL99], an optimization which merges non-contiguous I/O requests into contiguous ones at aggregator processes before sending them to file system (we have employed one aggregator). We have used ROMIO included in the MPICH-3.2 MPI distribution. Figures 5.10 and 5.11 shows VIDAS collective I/O and ROMIO collective I/O im- plementations for reading and writing, respectively. The experiments consists in collectively reading/writing data to/from an object/file of 1024 MB. The domains are accessing non-overlappingly interleaved strided vectors of 2MB blocks. We note that VIDAS collectives outperform collective I/O ROMIO operations by a considerable margin as we scale the number of virtual machines.

Figure 5.11 shows the results for VIDAS and ROMIO collective write implementations. NFS energy consumption is 7.5% to 107% higher than VIDAS, while VIDAS performance is 5% to 865% higher than NFS. PVFS energy consumption is 43% to 109% higher than VIDAS, while VIDAS performance is 17% to 636% higher than PVFS. Figure 5.10 shows VIDAS read performance up to 554% higher than ROMIO+NFS while NFS energy consumption is 65% higher. Similarly, VIDAS read performance is 420% higher than ROMIO+PVFS, while PVFS energy consumption is 61% higher.

The main explanation for the better performance of VIDAS is the shared collective buffer, which helps avoid copy operations. On the other hand, ROMIO collective operations copy the data into collective buffers before sending them to disks, which makes performance drop dramatically when increasing the number of virtual machines. In addition, since we always perform all I/O from host domain context, we are able to optimize data locality by placing shared memory objects on the same NUMA node as the I/O process which moves data to and from the storage device. This optimization increased our performance by about 10% and reduced performance variability. As the results of our VIDAS evaluation show, the reduction of data movement and improved performance also provides improved energy efficiency.

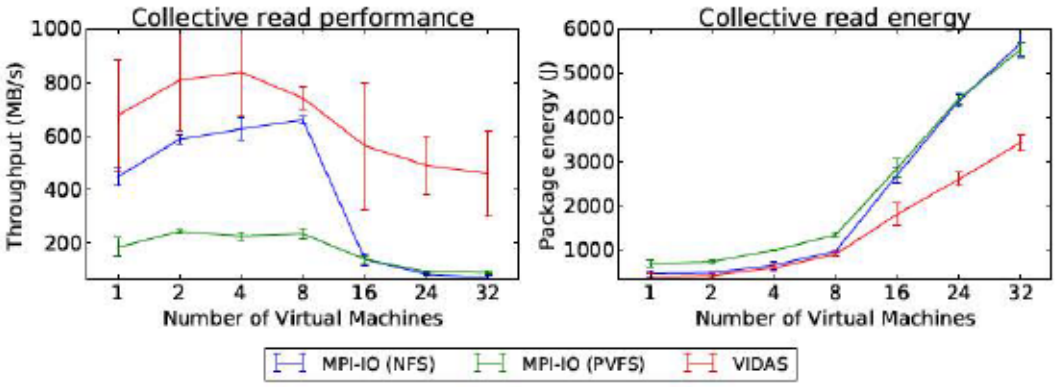


Figure 5.10: *Comparison of VIDAS collective read I/O and ROMIO collective read I/O.*

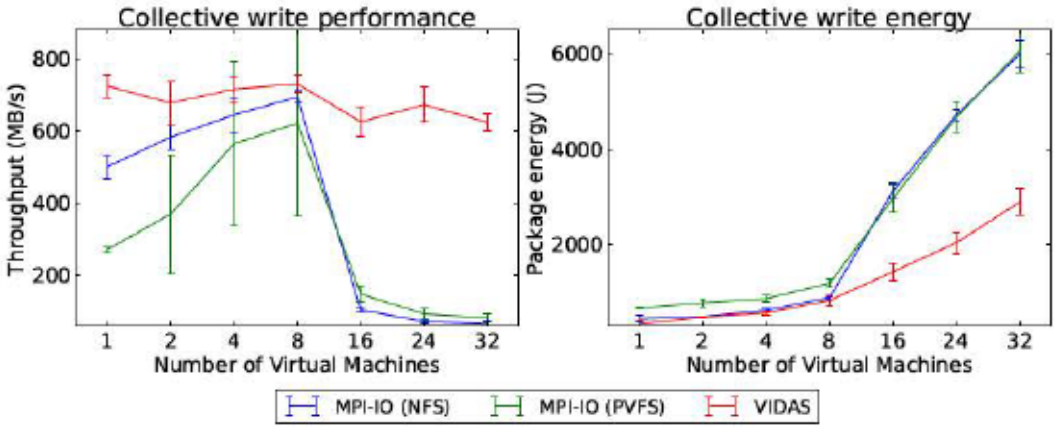


Figure 5.11: *Comparison of VIDAS collective write I/O and ROMIO collective write I/O.*

5.9 Summary

In this chapter we provide the design, implementation, and evaluation of VIDAS, an object-based virtualized data store that can be used to efficiently and consistently share access to externally stored data in virtualized environments based on a shared pool of storage objects. VIDAS provides integrated abstractions and mechanisms that allow to coordinate storage I/O across domains, create shared access spaces across node-local domains, relax the POSIX consistency, control the write and update policies, and control data locality. In order to efficiently implement VIDAS virtualized data sharing abstractions, we have proposed and evaluated a new data sharing mechanism which extends Xen to provide multi-domain memory sharing to user-space applications. The mechanisms and abstractions shown in this work would be best complemented with a higher-level layer which controls placement of VM machines, computing jobs, and data distribution. This would enable cloud-based HPC workloads to share data much more effectively by using VIDAS.

Chapter 6

Conclusions

In this thesis we have explored novel methodologies that improve the energy efficiency of data movement. Our work shows how increasing coordination between different layers of the operating system can have an impact on power and performance. In Chapter 3, we detail how the operating system's algorithms that coordinate and control the rate of data movement result in power and performance regimes, and use this to develop power prediction models. Chapter 4 shows a technique that improves energy efficiency by coordinating CPU performance with I/O regimes. Finally, Chapter 5 improves data sharing in virtualized environments by providing novel abstractions and mechanisms that allow efficient coordinated access to shared data objects.

The thesis properly fulfills all objectives presented in Section 1.2 in the following way.

***Understand*, *characterize*, and *model* energy consumed by data movement**. We have performed thorough instrumentation and analysis of power usage in the I/O stack across all system components in Sections 3.2 and 3.3, resulting in a methodology that relates system metrics to power usage in Section 3.4. Our findings enable the development of predictive power models in Sections 3.5 and 3.7. We also use this knowledge to develop an I/O-aware CPU management driver that is coordinated with I/O regimes in Section 4.4.

Explore novel CPU-level techniques for optimizing the performance and energy efficiency of data movement in the node-level I/O stack. We identify two main issues present in modern processors during I/O intensive workloads, thermal imbalances and inefficient use of CPU performance states. We exploit thermal imbalances to develop a more efficient thread placement policy for I/O and compute workloads in Section 4.3. We also apply the knowledge gathered from our analysis of power usage caused by I/O workloads to develop a more efficient, I/O-aware CPU performance driver in Section 4.4.

Research novel approaches for enhancing performance and energy efficiency of data sharing across virtualized domains. This thesis studies existing virtualized I/O stack solutions and introduces a new mechanism for improving virtualized data sharing based on efficient I/O forwarding in Section 5.1. We realize that the underlying data sharing mechanism of this solution can be improved upon by minimizing data movement and develop a new virtualized data sharing mechanism, VIDAS. Our solution, detailed in Section 5.5, greatly reduces data movement and provides enhanced programmability through novel abstractions, mechanisms and semantics that enable efficient inter-domain coordinated I/O.

6.1 Contributions

This thesis presents contributions to the study, analysis, and improvement of energy efficiency of data movement across different layers of the system. This thesis presents the following contributions.

Analysis and modeling of power consumption in the host I/O stack. We have performed thorough instrumentation and analysis of power usage in the I/O stack across all system components, revealing system transitions between power and performance regimes and operating system mechanisms and algorithms responsible for these regime changes. Our work also presents a methodology for automatically obtaining the system metrics that are most relevant for I/O power usage. We have proposed new analytical models that are able to predict power consumption of I/O workloads with various access patterns.

Micro-processor energy efficiency optimizations for I/O workloads. We have also conducted research on the impact of low-level systems software on the energy efficiency of the CPU during I/O. As a result, we propose two optimizations that improve the energy efficiency of the CPU through thermal and I/O-aware performance drivers.

Efficient data sharing in virtualized environments. This thesis introduces new abstractions and mechanisms that constitute a new virtualized data sharing solution. Our proposed solution improves performance and energy efficiency of intra-domain data sharing by minimizing data movement across the virtualized I/O stack using a novel data sharing mechanism. Our solution also improves programmability through new abstractions and consistency semantics that allow more expressive and efficient data sharing.

6.2 Thesis results

The main contributions of this thesis have been published in peer-reviewed conferences and journals. We list the publications derived from the work of this thesis. Seminars, posters, and awards are also included in this Section.

- Awards

  - *IBM PhD Fellowship Award*, 2013. The IBM Ph.D. Fellowship Awards Program is an intensely competitive worldwide program, which honors exceptional Ph.D. students who have an interest in solving problems that are important to IBM and fundamental to innovation in many academic disciplines and areas of study.
  - Best Poster Award for *Virtualized Data Sharing for High Performance Storage I/O* during ComplexHPC Spring School, Uppsala, 2013.

- Journals
  - Pablo Llopis, Javier Garcia Blas, Florin Isaila, and Jesus Carretero. *Survey of energy-efficient and power-proportional storage systems*. The Computer Journal, 2013.
  - Gabriel G. Castañé, Alberto Núñez, Pablo Llopis, and Jesús Carretero. *E-mc 2: a formal framework for energy modelling in cloud computing*. Simulation Modelling Practice and Theory 39, pp. 56-75, 2013.
  - Pablo Llopis, Manuel F. Dolz, Javier Garcia Blas, Florin Isaila, Mohammad Reza, and Michael Kuhn. *Analysing the Energy Consumption of the Storage Data Path*. Journal of Supercomputing, 2016.
  -

- Conferences
  - Pablo Llopis, Gonzalo Martin, Borja Bergua, and Jesus Carretero. *Virtual I/O forwarding for cloud-based HPC applications*. In Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, pp. 869-870. IEEE Computer Society, 2012.
  - Pablo Llopis, Gabriel G. Castañé, Jesús Carretero. *Cost-benefit analysis and exploration of cost-energy-performance trade-offs in scientific computing infrastructures*. In the Proceedings of the International Conference on Computational Science, 2016.

- Workshops
  - Pablo Llopis, Javier Blas, Florin Isaila, and Jesus Carretero. *VIDAS: object-based virtualized data sharing for high performance storage I/O*. In Proceedings of the 4th ACM workshop on Scientific cloud computing, pp. 37-44. ACM, 2013.
  - Pablo Llopis, Manuel F. Dolz, Javier García-Blas, Florin Isaila, Jesús Carretero, Mohammad Reza Heidari, and Michael Kuhn. "Analyzing Power Consumption of I/O Operations in HPC Applications." Ultrascale Computing Systems (NESUS 2015) Krakow, Poland (2015): 107.

- Posters
  - Pablo Llopis, Gonzalo Martin, and Jesus Carretero. *Virtual I/O forwarding for cloud-based HPC applications*. 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications.

  - Pablo Llopis, Javier Blas, Florin Isaila, and Jesus Carretero. *Virtualized Data Sharing for High Performance Storage I/O*. Best Poster. ComplexHPC Spring School 2013.

- Talks and Seminars

  - *Managing your infrastructure with code: An overview of automating configuration management with Puppet*, IBM Zurich Research Laboratory, Zurich, February 2014.
  - *Understanding the dynamic nature of modern processors in preparation for exascale computing*, Argonne National Laboratory, Chicago, July 2015.
  - *Work in progress about enhancing the programmability and energy efficiency of storage in HPC and cloud environments*, NESUS Winter School PhD Symposium, Timisoara, Romania, 2016.
  - *Thesis overview of Enhancing the programmability and energy efficiency of storage in HPC and cloud environments*, CAPAP-H Winter School, Madrid, Spain, 2016.

- Research Internships

  - IBM Zurich Research Laboratory, Zurich, Switzerland, 2013. Under the direction of host Ronald Luijten. Duration: 6 months.
  - Argonne National Laboratory, Chicago, USA, 2015. Under the direction of host Yoshii Kazutomo. Duration: 3 months.

6.3 Future directions

As a result of the work conducted for this thesis, we have identified several lines of research that could be pursued in the future.

6.3.1 Analysis and power modeling of I/O workloads

The thesis presents a thorough analysis of power usage in storage I/O and an analytical models, which are able to predict power consumption over various I/O workloads. We demonstrated this using micro-benchmarks and Filebench macro-benchmarks. In the future, we intend to expand this work to include more complex I/O workloads and to evaluate the accuracy of the model under a wider variety of scenarios, ranging from different memory technologies (such as NVRAM), to complex I/O access patterns. Our goal is to use this model as building blocks for modeling large distributed systems. In addition, we intend to develop an extension to the model that distinguishes data and metadata. A high-level abstraction which translates metadata operations such as file creation, file deletion, directory traversal, and file attribute access to power usage is useful for more accurately predicting energy consumption of certain workloads. However, given our experience with the models developed in this work, we think that this is a challenging problem due to the fact that metadata access will vary greatly between file systems and storage devices.

6.3.2 Virtualized data sharing

In this thesis, we presented two solutions for optimizing data sharing in virtual environments. In the future we plan to integrate VIDAS with our solution for I/O forwarding for cloud environments in order to combine node-local data sharing capabilities with high performance inter-node I/O delegation [LMBC12]. This approach would be useful for hierarchical data distribution policies based on reducing node-local communication and balancing storage I/O load over several nodes.

6.3.3 Energy efficient hardware device drivers

While the work in this thesis demonstrates two different optimizations for improving the energy efficiency of the CPU, we believe there are more opportunities that can be explored. Particularly concerning the increasing amount of dynamic features present in modern processors which lead to thermal issues. Opportunistic voltage and frequency scaling that takes advantage of the thermal headroom is increasingly being used for boosting performance. As the opportunistic frequency region becomes a greater portion of available frequencies, the dynamic behavior that is highly dependent on thermal and power limitations increases. The resulting dynamic behavior results in uneven heat accumulation and performance variation, leading a system with homogeneous processing units to behave as a heterogeneous system. This leaves the open question to how these dynamic features affect the efficiency and performance of I/O and compute workloads and how to deal with this dynamic behavior. In this thesis we start working on these issues but there is a wide range of scenarios and issues that need to be investigated.

Bibliography

[AAF+09] Miriam Allalouf, Yuriy Arbitman, Michael Factor, Ronen I Kat, Kalman Meth, and Dalit Naor. Storage modeling for power estimation. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, page 3. ACM, 2009.

[ACG+10] Hrishikesh Amur, James Cipar, Varun Gupta, Gregory R Ganger, Michael A Kozuch, and Karsten Schwan. Robust and flexible power-proportional storage. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 217–228. ACM, 2010.

[ACI+09] Nawab Ali, Philip Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert Ross, Lee Ward, and P Sadayappan. Scalable i/o forwarding framework for high-performance computing systems. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[AGSS11] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. HotOS, 2011.

[AS12] Hrishikesh Amur and Karsten Schwan. Achieving power-efficiency in clusters without distributed file system complexity. In *Computer Architecture*, pages 222–232. Springer, 2012.

[Axb] Jens Axboe. Flexible I/O tester. `http://freecode.com/projects/fio`.

[B+08] Richard Brown et al. Report to congress on server and data center energy efficiency: Public law 109-431. *Lawrence Berkeley National Laboratory*, 2008.

[Bar05] Luiz André Barroso. The price of performance. *Queue*, 3(7):48–53, 2005.

[BBC+13] S. Barrachina, M. Barreda, S. Catalán, M.F. Dolz, G. Fabregat, R. Mayo, and E.S. Quintana-Ortí. An integrated framework for power-performance analysis of parallel scientific workloads. In *ENERGY 2013, The 3rd International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, pages 114–119, 2013.

[BC11] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[BCH13] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[BF07] Cullen Bash and George Forman. Cool job allocation: Measuring the power savings of placing jobs at cooling-efficient locations in the data center. In *USENIX Annual Technical Conference*, volume 138, page 140, 2007.

[BH07] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *IEEE computer*, 40(12):33–37, 2007.

[BPSB00] Thomas D Burd, Trevor A Pering, Anthony J Stratakos, and Robert W Brodersen. A dynamic voltage scaled microprocessor system. *Solid-State Circuits, IEEE Journal of*, 35(11):1571–1580, 2000.

[BRPC08] Christian Belady, Andy Rawson, JOHN Pfleuger, and TAHIR Cader. Green grid data center power efficiency metrics: Pue and dcie. Technical report, Technical report, Green Grid, 2008.

[BSSG08] John S Bucy, Jiri Schindler, Steven W Schlosser, and Gregory R Ganger. The disksim simulation environment version 4.0 reference manual (cmu-pdl-08-101). *Parallel Data Laboratory*, page 26, 2008.

[BYFR+12] Muli Ben-Yehuda, Michael Factor, Eran Rom, Avishay Traeger, Eran Borovik, and Ben-Ami Yassour. Adding advanced storage controller functionality via low-overhead virtualization. In *FAST*, volume 12, pages 15–15, 2012.

[CAKLR11] Lauro Beltrao Costa, Samer Al-Kiswany, Raquel Vigolvino Lopes, and Matei Ripeanu. Assessing data deduplication trade-offs from an energy and performance perspective. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–6. IEEE, 2011.

[CBBA10] Andrea Castagnetti, Cécile Belleudy, Sébastien Bilavarn, and Michel Auguin. Power consumption modeling for dvfs exploitation. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 579–586. IEEE, 2010.

[CC07] Youngjin Cho and Naehyuck Chang. Energy-aware clock-frequency assignment in microprocessors and memory devices for dynamic voltage scaling. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(6):1030–1040, 2007.

[CCM+10] Adrian M Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K Gupta, Allan Snavely, and Steven Swanson. Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[CFKL95] Pei Cao, Edward W Felten, Anna R Karlin, and Kai Li. *A study of integrated prefetching and caching strategies*, volume 23. ACM, 1995.

[CG02] Dennis Colarelli and Dirk Grunwald. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Computer Society Press, 2002.

[CGF+10] Yanpei Chen, Archana Sulochana Ganapathi, Armando Fox, Randy H Katz, and David A Patterson. Statistical workloads for energy efficient mapreduce. *University of California at Berkeley, Technical Report No. UCB/EECS-2010-6*, 2010.

[CKR11] Jerry Chou, Jinoh Kim, and Doron Rotem. Energy-aware scheduling in disk storage systems. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 423–433. IEEE, 2011.

[CLGC14] Hung-Ching Chang, Bo Li, Matthew Grove, and Kirk W Cameron. How processor speedups can slow down i/o performance. In *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*, pages 395–404. IEEE, 2014.

[CM05] Gilberto Contreras and Margaret Martonosi. Power prediction for intel xscale® processors using performance monitoring unit events. In *Low Power Electronics and Design, 2005. ISLPED'05. Proceedings of the 2005 International Symposium on*, pages 221–226. IEEE, 2005.

[CPB03] Enrique V Carrera, Eduardo Pinheiro, and Ricardo Bianchini. Conserving disk energy in network servers. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 86–97. ACM, 2003.

[D+11] Jack Dongarra et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, page 1094342010391989, 2011.

[DHKF15] Manuel F. Dolz, Mohammad Reza Heidari, Michael Kuhn, and Germán Fabregat. ArduPower: a low-cost wattmeter to improve energy efficiency of HPC applications. In *5th International Green & Sustainable Computing Conference*, Las Vegas, NV, USA, December 2015.

[DKCC15] M. F. Dolz, J. Kunkel, K. Chasapis, and S. Catalán. An analytical methodology to derive power models based on hardware and software metrics. *Computer Science - Research and Development*, 2015.

[DoE14] US Department of Energy. Top Ten Exascale Research Challenges. Technical report, Department of Computer Science, Michigan State University, February 2014. `http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf`.

[DPNJ09] François Diakhaté, Marc Perache, Raymond Namyst, and Herve Jourdren. Efficient shared memory message passing for inter-vm communications. In *Euro-Par 2008 Workshops-Parallel Processing*, pages 53–62. Springer, 2009.

[ERKR06] Dimitris Economou, Suzanne Rivoire, Christos Kozyrakis, and Partha Ranganathan. Full-system power analysis and modeling for server environments. In *Workshop on Modeling Benchmarking and Simulation (MOBS)*, pages 13–23. Boston USA, 2006.

[FKL+91] Amos Fiat, Richard M Karp, Michael Luby, Lyle A McGeoch, Daniel D Sleator, and Neal E Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

[FWB07] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 13–23. ACM, 2007.

[GAH+12] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Eli: bare-metal performance for i/o virtualization. *ACM SIGARCH Computer Architecture News*, 40(1):411–422, 2012.

[GBG+10] Jorge Guerra, Wendy Belluomini, Joseph Glider, Karan Gupta, and Himabindu Pucha. Energy proportionality for storage: impact and feasibility. *ACM SIGOPS Operating Systems Review*, 44(1):35–39, 2010.

[Ge10] Rong Ge. Evaluating parallel i/o energy efficiency. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 213–220. IEEE Computer Society, 2010.

[GFS+10] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li, and Kirk W Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *Parallel and Distributed Systems, IEEE Transactions on*, 21(5):658–671, 2010.

[GHBD+09] Anshul Gandhi, Mor Harchol-Balter, Rajarshi Das, Jeffrey O Kephart, and Charles Lefurgy. Power capping via forced idleness. 2009.

[GMAC10] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A gpgpu transparent virtualization component for high performance computing clouds. In *Euro-Par 2010-Parallel Processing*, pages 379–391. Springer, 2010.

[GML+11] Francisco Giunta, Raffaele Montella, Giuliano Laccetti, Florin Isaila, and F Blas. A gpu accelerated high performance cloud computing infrastructure for grid computing based virtual environmental laboratory. *Advances in Grid Computing*, pages 121–146, 2011.

[GR11] John Gantz and David Reinsel. Extracting value from chaos. *IDC iview*, (1142):9–10, 2011.

[Gre15a] The Green Grid, 2015. http://www.thegreengrid.org/.

[Gre15b] The Green500 List, 2015. http://www.green500.org/lists/green201506.

[GSKF03] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mahmut Kandemir, and Hubertus Franke. Drpm: dynamic speed control for power management in server class disks. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 169–179. IEEE, 2003.

[GWW+10] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[GZS+03] Sudhanva Gurumurthi, Jianyong Zhang, Anand Sivasubramaniam, Mahmut Kandemir, Hubertus Franke, Narayanan Vijaykrishnan, and Mary Jane Irwin. Interplay of energy and performance for disk arrays running transaction processing workloads. In *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*, pages 123–132. IEEE, 2003.

[Hem10] Scott Hemmert. Green hpc: From nice to necessity. *Computing in Science and Engineering*, 12(6):8–10, 2010.

[HJL+08] Liting Hu, Hai Jin, Xiaofei Liao, Xianjie Xiong, and Haikun Liu. Magnet: A novel scheduling policy for power reduction in cluster with virtual machines. In *Cluster Computing, 2008 IEEE International Conference on*, pages 13–22. IEEE, 2008.

[HKGP07] Wei Huang, Matthew J Koop, Qi Gao, and Dhabaleswar K Panda. Virtual machine aware communication libraries for high performance computing. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 9. ACM, 2007.

[HSM+10] Brandon Heller, Srinivasan Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. Elastictree: Saving energy in data center networks. In *NSDI*, volume 10, pages 249–264, 2010.

[HSMR09] Stavros Harizopoulos, Mehul Shah, Justin Meza, and Parthasarathy Ranganathan. Energy efficiency: The new holy grail of data management systems research. *arXiv preprint arXiv:0909.1784*, 2009.

[ISSG05] Sandy Irani, Gaurav Singh, Sandeep K Shukla, and Rajesh K Gupta. An overview of the competitive and adversarial approaches to designing dynamic power management strategies. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 13(12):1349–1361, 2005.

[JDV+10] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Bruce Berriman, Benjamin P Berman, and Phil Maechling. Data sharing options for scientific workflows on amazon ec2. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–9. IEEE Computer Society, 2010.

[JS03] Satchit Jain and Siripong Sritanyaratana. Method and apparatus to implement the acpi (advanced configuration and power interface) c3 state in a rdram based system, October 14 2003. US Patent 6,633,987.

[JS08] Nikolai Joukov and Josef Sipek. Greenfs: Making enterprise computers greener by protecting them better. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 69–80. ACM, 2008.

[JVHLP10] Venkateswararao Jujjuri, Eric Van Hensbergen, Anthony Liguori, and Badari Pulavarty. Virtfs—a virtualization aware file system passthrough. In *Ottawa Linux Symposium (OLS)*, pages 109–120. Citeseer, 2010.

[KB10] Rini T Kaushik and Milind Bhandarkar. Greenhdfs: towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster. In *Proceedings of the USENIX Annual Technical Conference*, page 109, 2010.

[KBB+08] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, W Carson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, et al. Exascale computing study: Technology challenges in achieving exascale systems. 2008.

[KKJ+08] Kangho Kim, Cheiyol Kim, Sung-In Jung, Hyun-Sup Shin, and Jin-Soo Kim. Inter-domain socket communications supporting high performance and full binary compatibility on xen. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 11–20. ACM, 2008.

[KKK+08] Kelin Kuhn, Chris Kenyon, Avner Kornfeld, Mark Liu, Atul Maheshwari, Wei-kai Shih, Sam Sivakumar, Greg Taylor, Peter VanDerVoorn, and Keith Zawadzki. Managing process variation in intel's 45nm cmos technology. *Intel Technology Journal*, 12(2), 2008.

[KMA+11] Andrew Krioukov, Prashanth Mohan, Sara Alspaugh, Laura Keys, David Culler, and Randy Katz. Napsac: Design and implementation of a power-proportional web cluster. *ACM SIGCOMM computer communication review*, 41(1):102–108, 2011.

[KMKL12] Julian M Kunkel, Timo Minartz, Michael Kuhn, and Thomas Ludwig. Towards an energy-aware scientific i/o interface. *Computer Science-Research and Development*, pages 1–9, 2012.

[KMM12] Michael Knobloch, Bernd Mohr, and Timo Minartz. Determine energy-saving potential in wait-states of large-scale parallel programs. *Computer Science-Research and Development*, 27(4):255–263, 2012.

[Koo11] Jonathan Koomey. Growth in data center electricity use 2005 to 2010. *A report by Analytical Press, completed at the request of The New York Times*, 2011.

[LB13] Srinivas Pandruvada Len Brown, Jacob Pan. Power capping linux. LinuxCon, 2013.

[LBC+14] Jing Li, Anirudh Badam, Ranveer Chandra, Steven Swanson, Bruce L Worthington, and Qi Zhang. On the energy overhead of mobile storage systems. In *FAST*, pages 105–118, 2014.

[LBIC13] Pablo Llopis, Javier Garcia Blas, Florin Isaila, and Jesus Carretero. Survey of energy-efficient and power-proportional storage systems. *The Computer Journal*, page bxt058, 2013.

[LBMN09] Kien Le, Ricardo Bianchini, Margaret Martonosi, and Thu D Nguyen. Cost-and energy-aware load distribution across data centers. *Proceedings of HotPower*, pages 1–5, 2009.

[LDM01] Yung-Hsiang Lu and Giovanni De Micheli. Comparing system level power management policies. *Design & Test of Computers, IEEE*, 18(2):10–19, 2001.

[LGLZ12] Zhichao Li, Kevin M Greenan, Andrew W Leung, and Erez Zadok. Power consumption in enterprise-scale backup storage systems. *Power*, 2:2, 2012.

[LGT08] Adam Wade Lewis, Soumik Ghosh, and Nian-Feng Tzeng. Run-time energy consumption estimation based on workload in server systems. *HotPower*, 8:17–21, 2008.

[LHW12] Duy Le, Hai Huang, and Haining Wang. Understanding performance implications of nested file systems in a virtualized environment. In *FAST*, page 8, 2012.

[LJ03] Tao Li and Lizy Kurian John. Run-time modeling and estimation of operating system power consumption. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):160–171, 2003.

[LJSL09] Dingding Li, Hai Jin, Yingzhe Shao, and Xiaofei Liao. A high-efficient inter-domain data transferring system for virtual machines. In *Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication*, pages 385–390. ACM, 2009.

[LK10] Jacob Leverich and Christos Kozyrakis. On the energy (in) efficiency of hadoop clusters. *ACM SIGOPS Operating Systems Review*, 44(1):61–65, 2010.

[LMBC12] Pablo Llopis, Gonzalo Martin, Borja Bergua, and Jesus Carretero. Virtual i/o forwarding for cloud-based hpc applications. In *Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 869–870. IEEE Computer Society, 2012.

[LW04] Dong Li and Jun Wang. Eeraid: energy efficient redundant and inexpensive disk array. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 29. ACM, 2004.

[LWW08] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. Power capping: a prelude to power shifting. *Cluster Computing*, 11(2):183–195, 2008.

[LZKS09] Jay Lofstead, Fang Zheng, Scott Klasky, and Karsten Schwan. Adaptable, metadata rich io methods for portable high performance io. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.

[MAC+08] Dutch T Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J Feeley, Norman C Hutchinson, and Andrew Warfield. Parallax: virtual disks for virtual machines. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 41–54. ACM, 2008.

[MB06] Andreas Merkel and Frank Bellosa. Balancing power consumption in multiprocessor systems. *ACM SIGOPS Operating Systems Review*, 40(4):403–414, 2006.

[MCRS05] Justin D Moore, Jeffrey S Chase, Parthasarathy Ranganathan, and Ratnesh K Sharma. Making scheduling" cool": Temperature-aware workload placement in data centers. In *USENIX annual technical conference, General Track*, pages 61–75, 2005.

[MKL10] Timo Minartz, Julian M Kunkel, and Thomas Ludwig. Simulation of power consumption of energy efficient cluster hardware. *Computer Science-Research and Development*, 25(3-4):165–175, 2010.

[MMB13] Ioannis Manousakis, Manolis Marazakis, and Angelos Bilas. Fdio: A feedback driven controller for minimizing energy in i/o-intensive applications. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems, Berkeley, CA*, 2013.

[MST+05] Aravind Menon, Jose Renato Santos, Yoshio Turner, G John Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 13–23. ACM, 2005.

[Nat14] Natural Resources Defense Council. America's Data Centers Are Wasting Huge Amounts of Energy. Technical report, 2014. `www.nrdc.org/energy/files/data-center-efficiency-assessment-IB.pdf`.

[NDR08] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.

[NF04] Edmund B Nightingale and Jason Flinn. Energy-efficiency and storage flexibility in the blue file system. In *OSDI*, pages 363–378, 2004.

[NTD+09] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to ssds: analysis of tradeoffs. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 145–158. ACM, 2009.

[OAL14] Anne-Cecile Orgerie, Marcos Dias de Assuncao, and Laurent Lefevre. A survey on techniques for improving the energy efficiency of large-scale distributed systems. *ACM Computing Surveys (CSUR)*, 46(4):47, 2014.

[PB14] Eduardo Pinheiro and Ricardo Bianchini. Energy conservation techniques for disk array-based servers. In *25th Anniversary International Conference on Supercomputing Anniversary Volume*, pages 369–379. ACM, 2014.

[PFH+05] Ian Pratt, Keir Fraser, Steven Hand, Christian Limpach, Andrew Warfield, Dan Magenheimer, Jun Nakajima, and Asit Mallick. Xen 3.0 and the art of virtualization. In *Linux Symposium*, page 65. Ottawa, Ontario, Canada, 2005.

[PGC+13] Laura Prada, Javier Garcia, Alejandro Calderon, J Daniel Garcia, and Jesus Carretero. A novel black-box simulation model methodology for predicting performance and energy consumption in commodity storage devices. *Simulation Modelling Practice and Theory*, 34:48–63, 2013.

[PGGC11] Laura Prada, Javier Garcia, J Daniel Garcia, and Jesus Carretero. Power saving-aware prefetching for ssd-based systems. *The Journal of Supercomputing*, 58(3):323–331, 2011.

[PGR06] Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *NSDI*, 2006.

[PS04] Athanasios E Papathanasiou and Michael L Scott. Energy efficient prefetching and caching. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 255–268, 2004.

[PWB07] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *FAST*, volume 7, pages 17–23, 2007.

[RFB11] Ioan Raicu, Ian T Foster, and Pete Beckman. Making a case for distributed file systems at exascale. In *Proceedings of the third international workshop on Large-scale system and application performance*, pages 11–18. ACM, 2011.

[Rus08] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[SBP+05] Ratnesh K Sharma, Cullen E Bash, Chandrakant D Patel, Richard J Friedrich, and Jeffrey S Chase. Balance of power: Dynamic thermal management for internet data centers. *Internet Computing, IEEE*, 9(1):42–49, 2005.

[SGMV08] Mark W Storer, Kevin M Greenan, Ethan L Miller, and Kaladhar Voruganti. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, page 1. USENIX Association, 2008.

[Sha10] Jeffrey Shafer. I/o virtualization bottlenecks in cloud computing today. In *Proceedings of the 2nd conference on I/O virtualization*, pages 5–5. USENIX Association, 2010.

[SHM12] Robert Schöne, Daniel Hackenberg, and Daniel Molka. Memory performance at reduced cpu clock speeds: an analysis of current x86 64 processors. In *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*, pages 9–9. USENIX Association, 2012.

[SJC+14] Guangyu Sun, Yongsoo Joo, Yibo Chen, Yiran Chen, and Yuan Xie. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *Emerging Memory Technologies*, pages 51–77. Springer, 2014.

[SK06] Seung Woo Son and Mahmut Kandemir. Energy-aware data prefetching for multi-speed disks. In *Proceedings of the 3rd conference on Computing frontiers*, pages 105–114. ACM, 2006.

[SKK+90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *Computers, IEEE Transactions on*, 39(4):447–459, 1990.

[SKKJ07] Hyun-Sup Shin, Kang-Ho Kim, Chei-Yol Kim, and Sung-In Jung. The new approach for inter-communication between guest domains on virtual machine monitor. In *Computer and information sciences, 2007. iscis 2007. 22nd international symposium on*, pages 1–6. IEEE, 2007.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[SKZ08] Shekhar Srikantaiah, Aman Kansal, and Feng Zhao. Energy aware consolidation for cloud computing. In *Proceedings of the 2008 conference on Power aware computing and systems*, volume 10. San Diego, California, 2008.

[SL13] Georgia Sakellari and George Loukas. A survey of mathematical models, simulation approaches and testbeds used for research in cloud computing. *Simulation Modelling Practice and Theory*, 39:92–103, 2013.

[SNI12] SNIA - What's Old Is New Again - Tiered Storage, 2012. http://www.snia.org.

[TDN11] Eno Thereska, Austin Donnelly, and Dushyanth Narayanan. Sierra: practical power-proportionality for data center storage. In *Proceedings of the sixth conference on Computer systems*, pages 169–182. ACM, 2011.

[TGL99] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective i/o in romio. In *Frontiers of Massively Parallel Computation, 1999. Frontiers' 99. The Seventh Symposium on the*, pages 182–189. IEEE, 1999.

[THS10] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A Shah. Analyzing the energy efficiency of a database server. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 231–242. ACM, 2010.

[TWM+08] Niraj Tolia, Zhikui Wang, Manish Marwah, Cullen Bash, Parthasarathy Ranganathan, and Xiaoyun Zhu. Delivering energy proportionality with non energy-proportional systems-optimizing the ensemble. *HotPower*, 8:2–2, 2008.

[VAKC+12] Emalayan Vairavanathan, Samer Al-Kiswany, Lauro Beltrão Costa, Zhao Zhang, Daniel S Katz, Michael Wilde, and Matei Ripeanu. A workflow-aware storage system: An opportunity study. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 326–334. IEEE Computer Society, 2012.

[VKUR10] Akshat Verma, Ricardo Koller, Luis Useche, and Raju Rangaswami. Srcmap: Energy proportional storage using dynamic consolidation. In *FAST*, volume 10, pages 267–280, 2010.

[VSS+10] Arunchandar Vasan, Anand Sivasubramaniam, Vikrant Shimpi, T Sivabalan, and Rajesh Subbiah. Worth their watts?-an empirical study of datacenter servers. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–10. IEEE, 2010.

[Wil08] Andrew Wilson. The new and improved filebench. In *Proceedings of 6th USENIX Conference on File and Storage Technologies*, 2008.

[WPM02] Hang-Sheng Wang, Li-Shiuan Peh, and Sharad Malik. A power model for routers: Modeling alpha 21364 and infiniband routers. In *High Performance Interconnects, 2002. Proceedings. 10th Symposium on*, pages 21–27. IEEE, 2002.

[WR12] Carl Waldspurger and Mendel Rosenblum. I/o virtualization. *Communications of the ACM*, 55(1):66–73, 2012.

[Wu14] Fengguang Wu. Io-less dirty throttling. In *LinuxCon Japan 2012*. LinuxCon, 2014.

[WYZ08] Jun Wang, Xiaoyu Yao, and Huijun Zhu. Exploiting in-memory and on-disk redundancy to conserve energy in storage systems. *Computers, IEEE Transactions on*, 57(6):733–747, 2008.

[YZW08] Lamia Youseff, Dmitrii Zagorodnov, and Rich Wolski. Inter-os communication on highly parallel multi-core architectures, 2008.

[ZCT+05] Qingbo Zhu, Zhifeng Chen, Lin Tan, Yuanyuan Zhou, Kimberly Keeton, and John Wilkes. Hibernator: helping disk arrays sleep through the winter. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 177–190. ACM, 2005.

[ZDD+04] Qingbo Zhu, Francis M David, Christo Frank Devaraj, Zhenmin Li, Yuanyuan Zhou, and Pei Cao. Reducing energy consumption of disk storage using power-aware cache management. In *Software, IEE Proceedings-*, pages 118–118. IEEE, 2004.

[ZMRG07] Xiaolan Zhang, Suzanne McIntosh, Pankaj Rohatgi, and John Linwood Griffin. Xensocket: A high-throughput interdomain transport for virtual machines. In *Middleware 2007*, pages 184–203. Springer, 2007.

[ZOMM+15] Kaicheng Zhang, Seda Ogrenci-Memik, Gokhan Memik, Kazutomo Yoshii, Rajesh Sankaran, and Pete Beckman. Minimizing thermal variation across system components. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1139–1148. IEEE, 2015.

[ZSG+03] John Zedlewski, Sumeet Sobti, Nitin Garg, Fengzhou Zheng, Arvind Krishnamurthy, Randolph Y Wang, et al. Modeling hard-disk power consumption. In *FAST*, volume 3, pages 217–230, 2003.

[ZSZ04] Qingbo Zhu, Asim Shankar, and Yuanyuan Zhou. Pb-lru: a self-tuning power aware storage cache replacement algorithm for conserving disk energy. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 79–88. ACM, 2004.

[ZZ05] Qingbo Zhu and Yuanyuan Zhou. Power-aware storage cache management. *Computers, IEEE Transactions on*, 54(5):587–602, 2005.